ISSN 0265-2919
80p 31

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

An ORBIS Publication
IR £1 Aus $1.95 NZ $2.25 SA R1.95 Sing $4.50 USA & Can $1.95

CONTENTS

Next Week

- **The programming techniques series is succeeded by a new series on integrated software. We review the market leaders such as Symphony and Framework.**
- **Our robotics series moves on to consider feedback control in robot movement — how does a robot follow the straight and narrow?**

QUIZ

1) What was 'Elektro'?
2) What are 'stubs'?
3) What is a 'high-pass filter'?
4) What does 'CMOS' stand for, and what is its significance?

Answers To Last Week's Quiz

1) An array is dynamically dimensioned when its extent in the DIM statement is defined at run time by some input variable, rather than being a predetermined absolute number, for example: DIM A$(X,Y).

2) NOWRAP is a LOGO command that causes the screen turtle to halt at the screen boundary, rather than 'wrapping-round' to the opposite edge of the screen.

3) On the Spectrum, LET X=USR "A" returns the start address of the eight-byte table defining the first user-defined graphics character.

4) When the output of a system is fed back into the system to control the inputs, this is called feedback control.

COVER PHOTOGRAPH BY IAN McKINNELL; DALEK AND ROBOT KINDLY LOANED BY FORBIDDEN PLANET, 58 ST GILES HIGH ST, LONDON WC2; FURNITURE COURTESY OF PRACTICAL STYLING AT CENTREPOINT, LONDON WC2

Editor Mike Wesley; **Art Director** David Whelan; **Technical Editor** Brian Morris; **Production Editor** Catherine Cardwell; **Art Editor** Claudia Zeff; **Chief Sub Editor** Robert Pickering; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Editorial Assistant** Stephen Malone; **Sub Editor** Steve Mann; **Researcher** Melanie Davis; **Staff Writer** Steve Colwill; **Contributors** Geoff Bains, Harvey Mellor, Mike Curtis, Steve Colwill, Chris Naylor, Graham Storrs; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editors** Chris Cooper, Maurice Geller; **Production Controller** Peter Taylor-Medhurst; **Circulation Director** David Breed; **Marketing Director** Michael Joyce; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; © **APSIF Copenhagen 1984; © Orbis Publishing Ltd 1984: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Artisan Press Ltd, Leicester**

HOME COMPUTER ADVANCED COURSE – Price UK 80p IR £1.00 AUS $1.95 NZ $2.25 SA R1.95 SINGAPORE $4.50 USA and CANADA $1.95

How to obtain your copies of HOME COMPUTER ADVANCED COURSE – Copies are obtainable by placing a regular order at your newsagent, or by taking out a subscription. Subscription rates: for six months (26 issues) £23.80; for one year (52 issues) £47.60. Send your order and remittance to Punch Subscription Services, Watling Street, Bletchley, Milton Keynes, Bucks MK2 2BW, being sure to state the number of the first issue required.

Back Numbers UK and Eire – Back numbers are obtainable from your newsagent or from HOME COMPUTER ADVANCED COURSE. Back numbers, Orbis Publishing Limited, 20/22 Bedfordbury, LONDON WC2N 4BT at cover price. AUSTRALIA: Back numbers are obtainable from HOME COMPUTER ADVANCED COURSE. Back numbers, Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G Melbourne, Vic 3001. SOUTH AFRICA, NEW ZEALAND, EUROPE & MALTA: Back numbers are available at cover price from your newsagent. In case of difficulty write to the address in your country given for binders. South African readers should add sales tax.

How to obtain binders for HOME COMPUTER ADVANCED COURSE – UK and Eire: Please send £3.95 per binder if you do not wish to take advantage of our special offer detailed in Issues 5, 6 and 7. EUROPE: Write with remittance of £5.00 per binder (incl. p&p) payable to Orbis Publishing Limited, 20/22 Bedfordbury, LONDON WC2N 4BT. MALTA: Binders are obtainable through your local newsagent price £3.95. In case of difficulty write to HOME COMPUTER ADVANCED COURSE BINDERS, Miller (Malta) Ltd, M.A. Vassalli Street, Valletta, Malta. AUSTRALIA: For details of how to obtain your binders see inserts in early issues or write to HOME COMPUTER ADVANCED COURSE BINDERS, First Post Pty Ltd, 23 Chandos Street, St. Leonards, NSW 2065. The binders supplied are those illustrated in the magazine. NEW ZEALAND: Binders are available through your local newsagent or from HOME COMPUTER ADVANCED COURSE BINDERS, Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington. SOUTH AFRICA: Binders are available through any branch of Central Newsagency. In case of difficulty write to HOME COMPUTER ADVANCED COURSE BINDERS, Intermag, PO Box 57394, Springfield 2137.

Note – Binders and back numbers are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK market only and may not necessarily be identical to binders produced for sale outside the UK. Binders and issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.

DUCKS & DROIDS

Machines created in the image of man that can perform human tasks have long fascinated engineers and writers alike. We begin a series of articles that explore the science of robotics, starting here with a look at past attempts to build such machines, from the mechanical inventions of the 18th century to present day industrial robots.

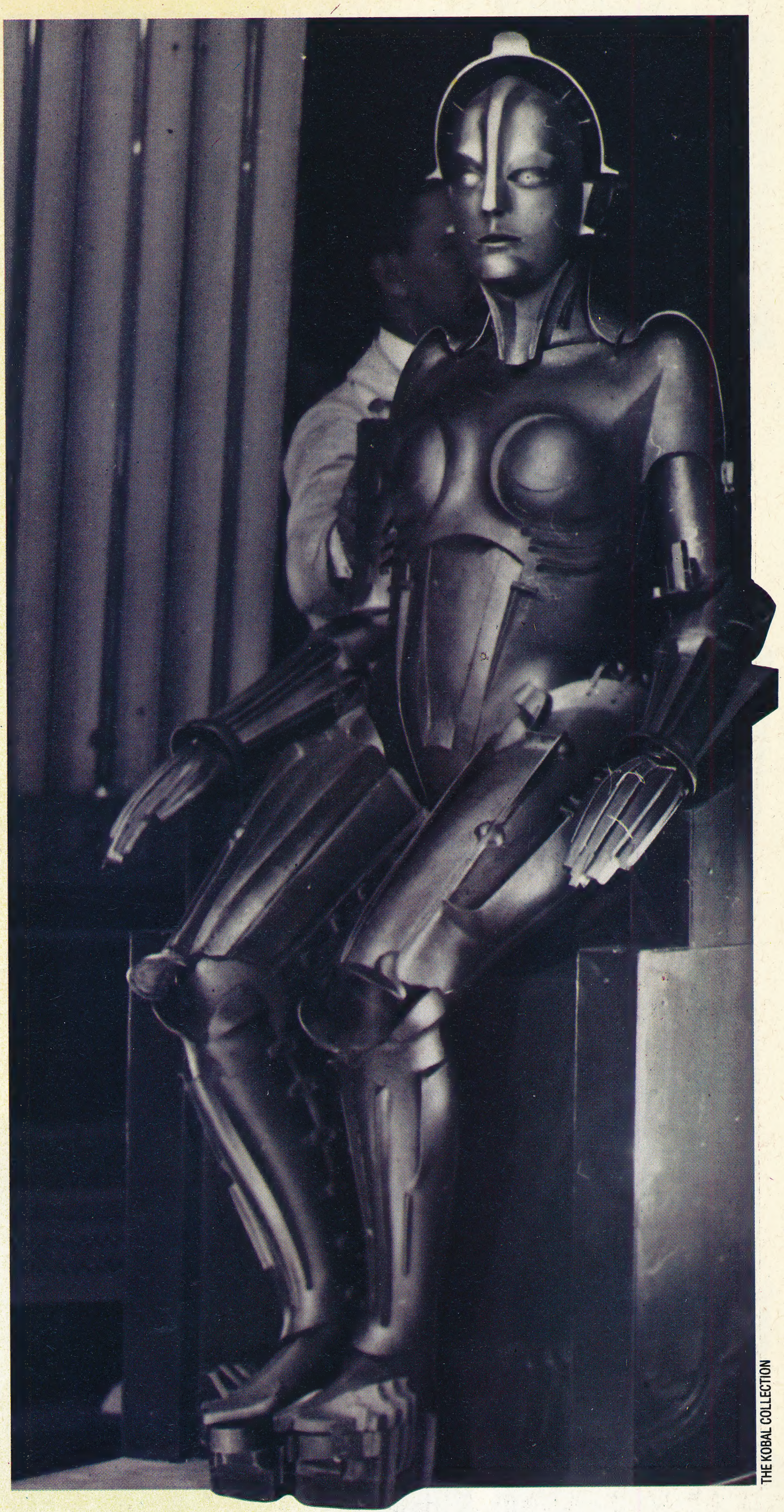
For hundreds of years, many people have been attracted to the idea of mechanical men in one form or another. Philosophers, engineers and inventors have set their minds to creating machines that mimic human behaviour. Although robots are nowadays less likely to look like humans and are designed to perform a specific range of actions, the earliest mechanical men were designed to look as lifelike as possible and suggest that they could be capable of any human action.

The first mechanical robot, however, wasn't given the form of a man. In 1738, Jacques de Vaucanson (1709–1782), a French engineer, presented a mechanical duck to the Académie Royale des Sciences in Paris. The duck was able to flap its wings, quack and eat food. Later in the 18th century, a Swiss inventor, Pierre Jacquet-Droz (1721–1790), created a set of mechanical puppets that could perform a variety of actions. One could write, one was able to draw figures, and another played music on an organ. By the end of the 19th century there were large numbers of such machines in existence, all based on clockwork mechanisms.

In the Victorian era, numerous figures were constructed that were remarkably lifelike, and not all of these were based on clockwork. In 1893, George Moore built a mechanical man that relied on steam power as its motivating force — an interesting side-effect of which was to enable the mechanical man to puff a cigar and appear to exhale smoke.

Newer technologies have stimulated the development of more ambitious machines: from the simple mechanical men constructed out of Meccano sets that are capable of walking across the floor, to the classic 'Elektro' man, built by the American company, Westinghouse. 'Elektro' was a seven foot tall (2.15 metres) mechanical man that could speak up to 80 different words, count, walk, talk, salute and distinguish between different colours. He was powered by no less than 11 electric motors and weighed 117 kgs (260 lbs). Controlling this huge bulk was a 'brain' that consisted of a total of 82 different relays.

But each of these mechanical men had its limitations. None of them, despite their obvious entertainment value, had any of the capabilities you might want when you think of your ideal robot. A mechanical man that can draw figures will not do the shopping for you, and a mechanical man that can walk across the room will be unlikely to get even as far as the shops without walking into


THE KOBAL COLLECTION

Robot Roots
Fritz Lang's 1926 science fiction classic 'Metropolis' influenced film-makers and audiences for decades, not least in crystallising vague contemporary images of progress and industry in The Machine, the first cinema robot star

a lamppost. Each of these mechanical men was very definitely a machine — they typically performed a limited range of actions that required no decisions to be made, and did not appear to embody any kind of intelligence.

ROBOTS IN FICTION

But if the inventors and engineers were stuck for ideas, the writers of fiction certainly didn't feel these creative restrictions. Science fiction has thrived on the idea of robots. In fact, the very word *robot* is the product of a fictional work. In 1923, the Czech playwright, Karel Čapek (1789–1938) wrote a play called *R.U.R.* — the title was an abbreviation for Rossum's Universal Robots. The play was about the invention of mechanical men so perfect that they could carry out all of the tasks a human might perform. Eventually, the robots found that they had no use for men at all, which left the humans in a rather precarious position. In Czech, the word *robota* simply means 'worker'. So, the title of Čapek's play should have been translated as 'Rossum's Universal Workers' but, somehow, the word 'robot' caught on, and this has since become the standard term for any mechanical man with human capabilities.

Fictional fantasies about creatures constructed to resemble human beings go back to Mary Shelley's well-known Gothic novel, *Frankenstein* (1818). Although it was not mechanical, the monster created by Victor Frankenstein was constructed from a set of parts, even though these were obtained by the rather gruesome process of raiding graveyards. The invading creatures in H.G. Wells' *The War Of The Worlds* (1898) were, at least in part, robotic.

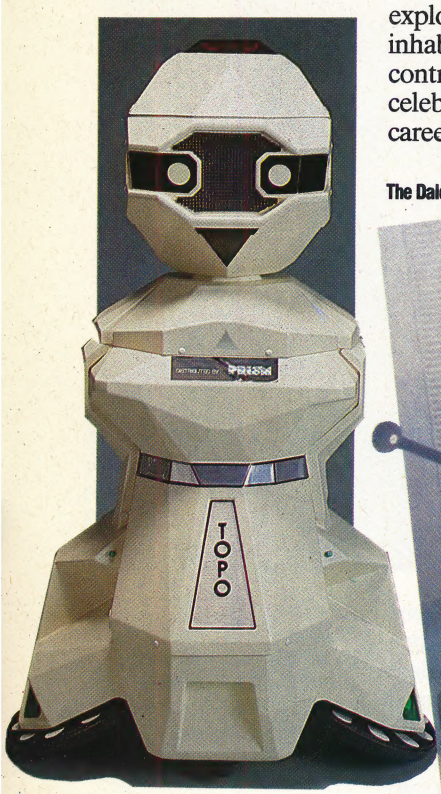
Novelists of the 20th century, however, have explored in immense detail a fictional world inhabited by robots. The most notable contribution has been that of Isaac Asimov, the celebrated science fiction writer who began his career in 1940 writing short stories about robots and their imagined operational problems. So complete is Asimov's visionary robotic world that he has even formulated the three Laws of Robotics. According to Asimov, the Laws are contained in the *Handbook Of Robotics* (56th Edition, 2058 AD). Clearly, he was allowing a very reasonable timescale before robots become commonplace.

In the cinema and on television, robots have also made their fictional mark. The television series *Dr Who* is densely populated with Daleks and Cybermen, and in the *Star Wars* films C3PO and R2D2 are the equals of their human co-stars.

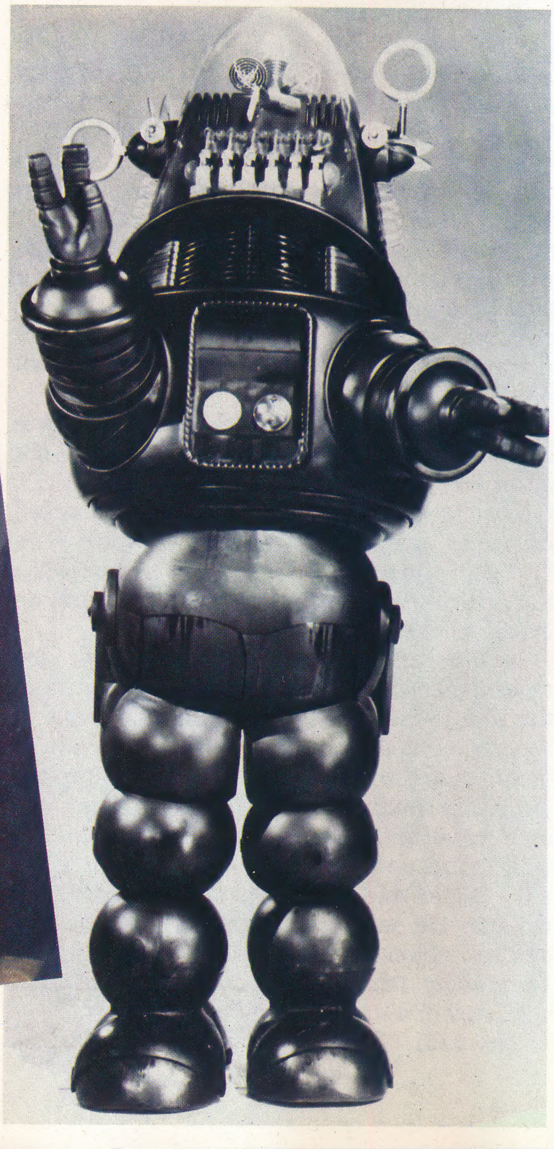
In comparison with these flights of fantasy, the present day use of robots seems quite mundane. The industrial robots found on car assembly lines receive most of the attention nowadays. It is estimated that by 1985 there will be 25,000 robots in use in Japanese industry, 15,000 in the USA and 8,000 in West Germany. Britain's robot population is among the smallest of any industrial nation. By 1985, only 1,500 robots are expected to be in operation. Expansion of the European market for industrial robots is expected to continue unabated: by 1990 it is estimated that it will be worth £350 million.

But, for many people, industrial robots seem rather dull. A machine that repeatedly welds parts

Fact And Fiction
The most famous robots on television must be the Daleks. These are really armoured personnel carriers, controlled by their creators riding inside.
Robbie The Robot from the film 'Forbidden Planet', epitomises the caring, sensitive robot of anthropomorphic legend.
Topo, Prism's now-discontinued household robot, was a half-serious attempt to introduce robotics into the home



The Daleks



Robbie The Robot

on the framework of a car, or endlessly sprays paint, is hardly the fictional image of robots. Whether the ideal robot will ever be created is a matter of conjecture. And whether such a robot would be designed in the image of man is also difficult to determine. But by taking a close look at some aspect of robotics, as we will be doing in this series of articles, we can judge for ourselves what form the robots of the future may have.

Robot Speak

Robots, at least in fiction, have been given such a variety of names that you might find it useful to have a glossary of the most common terms used. Bear in mind though that just because something is given a name, it does not necessarily imply that it actually exists!

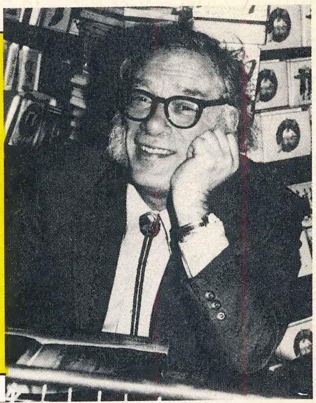
Android: A robot designed to look like a human being in every respect.
Anthropomorphic: Literally 'man-like'. An android is anthropomorphic in every respect but many robots are designed to be anthropomorphic in only some respects. For example, they may have an arm that is like a human arm.
Automation: The automatic control of a manufacturing process.
Automaton: A machine with concealed workings that usually performs only a predetermined series of functions. The early mechanical men were automata. It also has a more technical meaning when associated with *automata theory*, which is an analytical system by which any device can be studied and described — robots, computers, even people.
Cybernetics: The study of systems of control and communications. Devised by Norbert Weiner in 1947, the central claim of cybernetics is that it can be used to examine biological systems as if they were machines.
Cybert: A fictional idea of a mechanical humanoid.
Cybot: Also fictional; a robot with human mental abilities.
Cyborg: A *Cyb*ernetic *org*anism in which some parts are biological and others mechanical.
Doppelgänger: An exact replica of a particular living person — although this is usually a spirit or ghost.
Droid: A good robot that obeys Asimov's Three Laws.
End effector: Current terminology for a robot's 'hand'.
Homunculi: Little men or manikins.
Manipulator: Another term for a robot's 'hand'.
Mechanisation: The replacement of one process by a mechanical process.
Metal-collar workers: Industrial robots. Human office workers are often called 'white-collar' workers, and manual workers are known as 'blue-collar' workers. Inevitably, robots have started to be referred to as metal-collar workers.
Robot: A machine that is able to carry out some human functions, although it may not necessarily look particularly human.
Robotics: The science of studying robots.

Asimov's Laws Of Robotics

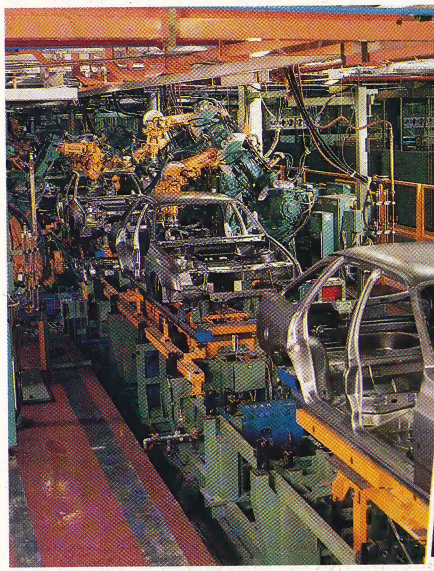
1. A robot may not injure a human being, or, through inaction allow a human being to come to harm.

2. A robot must obey the orders given to it by human beings except where such orders would conflict with the First Law.

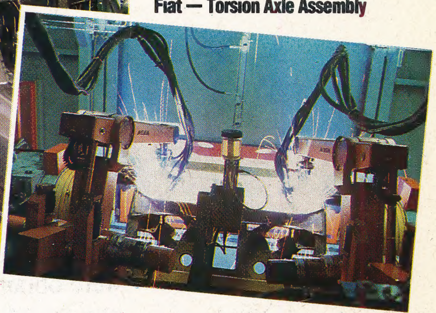
3. A robot must protect its own existence as long as such protection does not conflict with the First and Second Laws.



Ruling The Robot
When robots are capable of independent action, then Asimov's Laws may well form the basis of their behaviour. Today's robots, however, are incapable of identifying a human, so the mores of robot-human interaction are as yet irrelevant



Ford — The Sierra Assembly Line



Fiat — Torsion Axle Assembly

Robot Assembly
For some time to come, robots will be used mainly on production lines. The economics of mass production make them ideal assembly-line workers, as the Ford and Fiat factories plainly show. Specialisation usually demands that these robots be reduced to one or two arms equipped with grippers, spanners and welding gear

What Is A Robot?

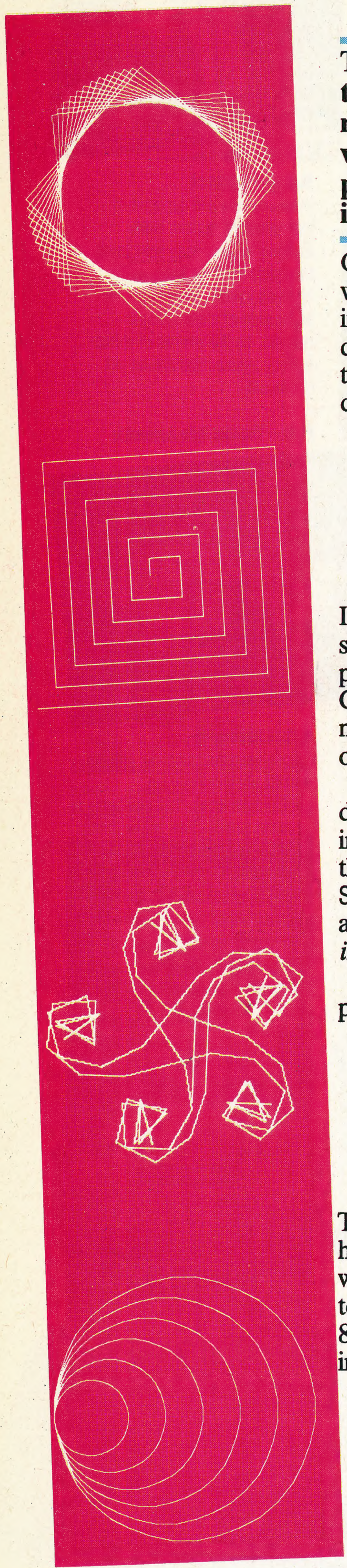
If you look in the glossary you will see that we have defined a robot as 'a machine that is able to carry out some human functions, although it may not look particularly human'. Obviously, this is a very wide definition — it could be applied, for instance, to computers (because they carry out some human calculating functions). In common usage, however, a robot should have recognisable human qualities. It may be able to move around, or perhaps even walk. It may have an arm that resembles a human arm. It may be able to see things and hear things. It may even have a very high degree of intelligence.

The exact form and capabilities of robots depend in the main on two things: what we want them to do, and what we can get them to do. For instance, an industrial robot used for welding may not be able to move around — not because we could not make a robot move around, but because we want it to stay in one place and get on with the welding. Similarly, a domestic robot may be able to make a cup of tea, but it might not be able to bring it upstairs to your bed, because it may not be possible to build a robot that can climb stairs without spilling your tea!

The term 'robot' has become the generic word for all human-like machines, and the limitations on what they are and what they can do rest with those who design and build them. These limits are being expanded almost daily.

REPEAT PERFORMANCE

The LOGO language uses the mathematical technique of recursion (an instruction that refers to itself) to great effect. Coupled with variable inputs, the use of recursion in procedures can produce some extremely interesting results.



One of the first programs we defined in the course was a procedure to draw a square. The definition instructed the turtle to move forward a certain distance, turn right 90 degrees and repeat those two steps three more times. Here is another way to draw a square:

```
TO SQUARE
   FD 50
   RT 90
   SQUARE
END
```

If you were to try this out, the turtle would draw a square and then carry on moving around the perimeter of the square until you pressed Control-G or BREAK. The most noticeable thing about this new SQUARE procedure is that it calls itself — in other words, it is 'recursive'.

When this procedure is run, LOGO fetches the definition of SQUARE and begins to obey the instructions. The turtle is moved FORWARD 50 and then turned RIGHT 90. The next instruction is SQUARE, so LOGO fetches the definition of SQUARE and begins to obey it. This will go on *ad infinitum* if the program is not interrupted.

It is also possible to use recursive calls in procedures that require inputs:

```
TO POLY :SIDE :ANGLE
   FD :SIDE
   RT :ANGLE
   POLY :SIDE :ANGLE
END
```

This procedure can produce all the polygons we have defined so far in the course (see page 545) as well as many we haven't looked at (you might like to try using the procedure with an angle value of 89). It is also possible to change the value of the input in the recursive call. Thus:

```
TO POLYSPI :SIDE :ANGLE
   FD :SIDE
   RT :ANGLE
   POLYSPI ( :SIDE + 5 ) :ANGLE
END
```

The only difference between this procedure and POLY is that five is added to the value of SIDE each time it is called. So if you began with POLYSPI 10 90, then the first call would draw a line of length 10, the second would be 15, then 20, and so on. The result is a spiral. You might like to experiment with different inputs: 10 90, 10 95, 10 120, 10 117, 10 144 and 10 142 are interesting starters. You could also try modifying the procedure — one possibility is to change addition to subtraction or multiplication.

Here's a similar procedure that increments the angle rather than the side value:

```
TO INSPI :SIDE :ANGLE :INC
   FD :SIDE
   RT :ANGLE
   INSPI :SIDE ( :ANGLE + :INC ) :INC
END
```

Try various inputs: 5 0 7, 10 40 30, 15 2 20, 5 30 20 will do initially. Why do some shapes close and others not? Can you find a rule?

The simple repetition of a piece of code is referred to as *iteration.* LOGO uses REPEAT for this purpose, while other languages use a variety of constructs, such as FOR . . . NEXT, REPEAT . . . UNTIL, and WHILE . . . WEND. However, LOGO relies much more on recursion than it does on iteration. If you've programmed in other languages you may have difficulty in breaking away from using iteration, but turtle graphics is ideal for experimenting with recursive calls.

STOP RULES

All of the recursive procedures we have looked at so far continue repeating indefinitely. Clearly, we need a way to make a procedure stop at some point. Taking the SQUARE procedure as our example, a possible place to stop it would be after it has drawn a complete square and the turtle's heading is back to 0. This can be done by adding a 'stop rule' to the procedure:

```
TO SQUARE :SIDE
   FD :SIDE
   RT 90
   IF HEADING = 0 THEN STOP
   SQUARE :SIDE
END
```

The new primitives are STOP and IF. The first of these commands causes a procedure to stop running and returns control to the calling procedure. An IF statement is LOGO's way of making decisions. IF is followed by a condition, and THEN by an action that is carried out if the condition is true.

Let's look at a version of POLYSPI with a stop rule and consider exactly what happens when it is run:

```
TO POLYSPI : LENGTH
   IF :LENGTH > 15 THEN STOP
   FD :LENGTH
   RT 90
   POLYSPI ( :LENGTH + 5 )
END
```

This is what happens when we run POLYSPI 10. The POLYSPI procedure is called and a local variable is defined with its value set at 10. Since this value is not greater than 15, LOGO proceeds to carry out the movement FD 10 RT 90, and then makes a new call to POLYSPI, but this time with an input value of 15. This causes a copy of the procedure to be called again. Because LENGTH is not greater than 15, the turtle is made to move FD 15 RT 90, and another call to POLYSPI is made. But this time, the local variable has been increased to 20, so the procedure stops and returns control to the procedure that called it (POLYSPI 15). This procedure in turn has come to its final line, and returns control to its calling procedure. This also stops, at which point the program has come to its end.

We have shown how recursion in LOGO involves procedures calling copies of themselves. It is important to keep in mind that the recursive calls are copies that exist alongside the original procedure, working as if they were completely different from it. When finished, such a procedure always returns control to the procedure that called it. To illustrate more clearly the process of returning from procedure calls, we can rearrange POLYSPI in this way:

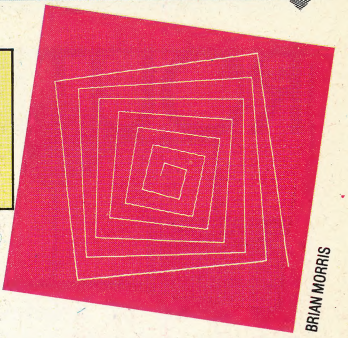
```
TO POLYSPI :LENGTH
   IF :LENGTH > 15 THEN STOP
   POLYSPI ( :LENGTH + 15 )
   FD :LENGTH
   RT 90
END
```

If you run this you will see that the program does its drawing 'backwards': the lines are drawn spiralling inwards rather than outwards. (This will be shown more clearly if you use a larger value in the condition statement — for example, using 50 instead of 15.) What is significant here is that LOGO draws each line as control is returned from the procedure calls. In our previous example, a line was drawn and control was then passed to another procedure. But here, all the procedures are called before any drawing begins, and the last created value of LENGTH is the one used first.

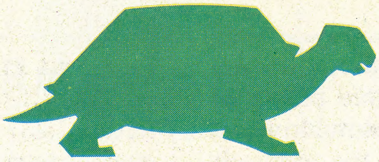
Finally, we should note that recursion is a technique that uses up a lot of memory. Procedures in which the recursive call is in the last line are the most efficiently implemented, however, as they don't take up any extra memory no matter how many times they're called. If a procedure can be written so it is 'end recursive' then this is usually worth doing.

Procedure Problem 3

Write a recursive procedure to draw a tower of squares one on top of the other, halving the length of the side each time.



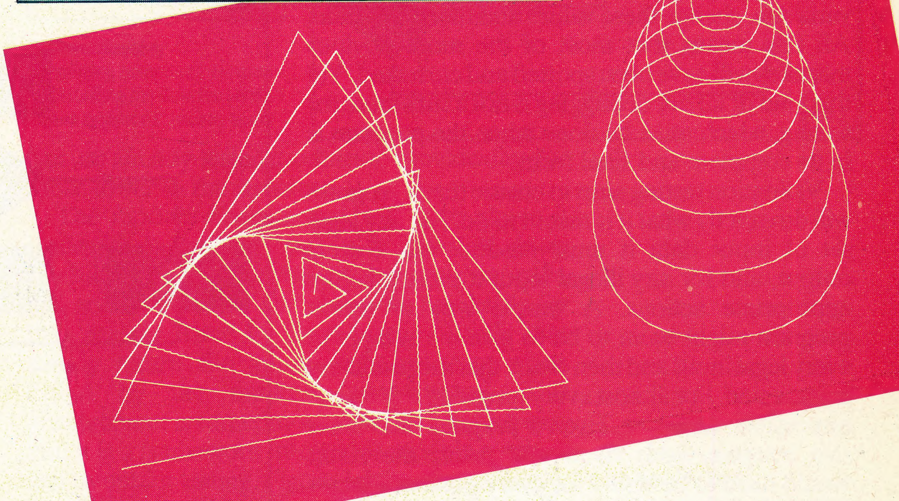
BRIAN MORRIS



Logo Flavours

In all LCSI versions the action part of an IF statement is written as a list within square brackets and without the THEN. For example:

```
IF HEADING = 0 [STOP]
```



Exercise Answers

A procedure to draw a circle, given the radius as input, with the present position as a point on the circumference:

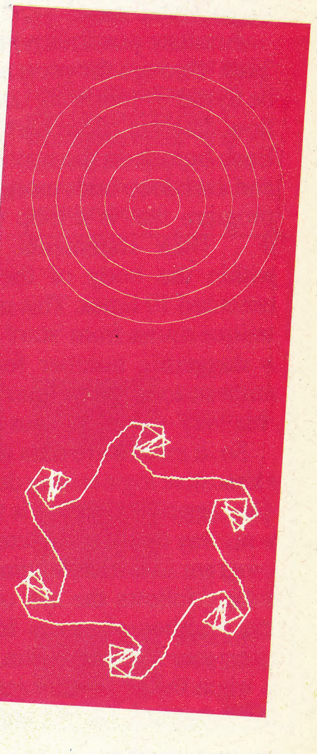
```
TO CIRCLE :RADIUS
   REPEAT 36 [FD ( 2 * :PI * :RADIUS / 36 )
   RT  10]
END
MAKE "PI (3.14159)
```

If we adapt this procedure so that the centre of the circle is at the present position we get:

```
TO C.CIRCLE :RADIUS
   PU LT 90 FD :RADIUS RT 90 PD
   CIRCLE :RADIUS
   PU LT 90 BK :RADIUS RT 90 PD
END
```

TARGET uses C.CIRCLE to draw 5 concentric circles:

```
TO TARGET
   C.CIRCLE 10 C.CIRCLE 20 C.CIRCLE 30
   C.CIRCLE 40 C.CIRCLE 50
END
```



ROUTINE CHECK UP

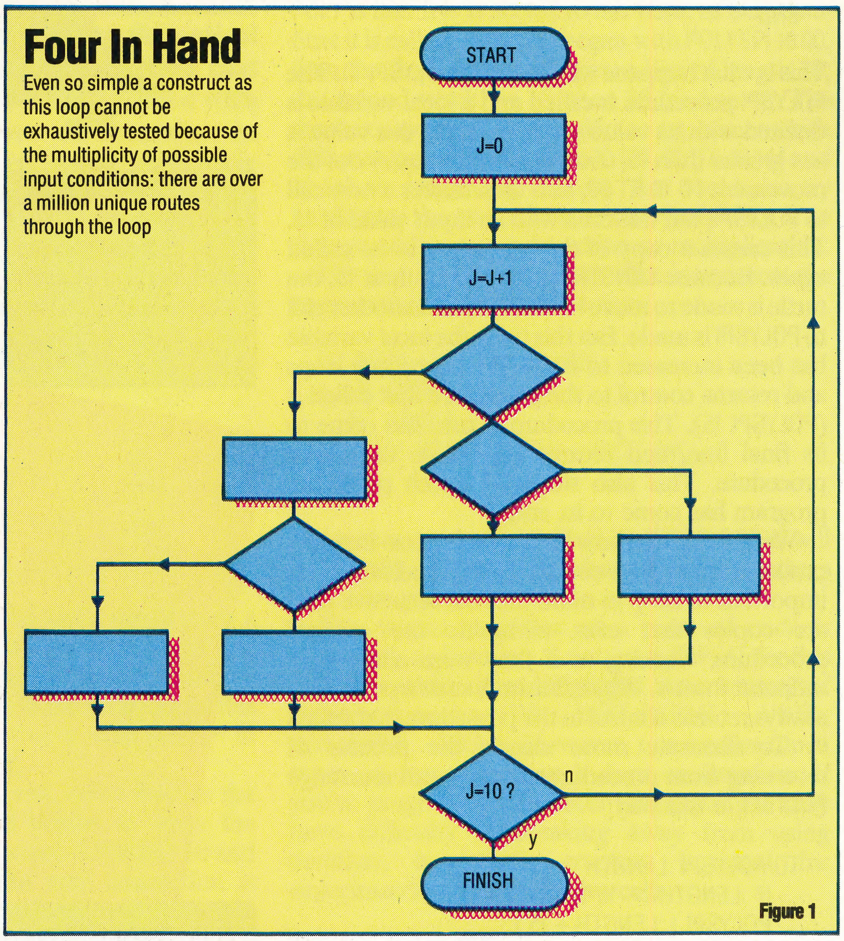
Our series of articles on programming techniques should have provided plenty of ideas for program design and development. In this final part, we discuss the methods that may be used to test a finished program.

One of the great advantages of programming in an interpreted language like BASIC is that code can be tested as it is being written. The programmer can, at any time, type RUN and see what happens. On most machines, it is a simple matter to 'break' into a running program, PRINT the values of key variables, change these values and then CONTinue. All this means that most of the more obvious mistakes will have been spotted and corrected. Yet this kind of *ad hoc* debugging is not a substitute for testing, which must be done when the program is in its complete and final form.

Validation testing aims to ensure that a program will do exactly what it is meant to do. For any legal set of input data it must produce the correct output, and for any illegal input it must take the appropriate actions. A simple way to test a program might seem to be to give it a sample of every legal input and then check that the results are as expected. For almost every program, this will be impossible, however. Even a program that takes two integers, adds them and prints the result would need to be tested for every possible integer value! Yet this is only part of the problem, as every *illegal* value would need to be tested, too.

Another possibility might be to look at every 'path' through the program. A particular path can be found by following one route through a control flow diagram (flowchart) from beginning to end. Each branch on the way allows for alternate paths and each loop adds more. Figure 1 shows a simple program that is a loop containing a number of IF...THEN statements. There are four paths within the body of the loop and the loop is executed 10 times. This means that the number of unique routes from 'start' to 'finish' is 1,398,100 — a staggering number for what would probably amount to a dozen lines of code. Clearly, testing this way would be out of the question.

So, if exhaustive data testing does not work and exhaustive logic testing does not work, what does? The surprising answer is that nothing does. There is no way to test completely a reasonably complex program in a realistic time. Partly for this reason, testing follows the law of diminishing returns — the number of errors found per unit of effort decreases with each extra unit. So, the time to stop is when the effort of doing it outweighs the cost of the program's (as yet undetected) faults.



Four In Hand

Even so simple a construct as this loop cannot be exhaustively tested because of the multiplicity of possible input conditions: there are over a million unique routes through the loop

Figure 1

However, despite these drawbacks, it is worth devising some method of testing. A reasonable assumption is that if a machine will operate correctly on one datum of a particular type it will operate correctly on all data of the same type. So, if a subroutine works for one positive integer within its range, it should work for all positive integers in that range. This leads us to a type of testing known as 'equivalence class testing'. The idea is to develop a set of test cases that are each representative of a class of cases that should all behave in the same way. Thus, if a piece of code checks that an input is in the range 1 to 100, we should test for inputs that are less than the lowest value expected, greater than the highest value, and within the expected range (value<1; value>100; and 1=<value=<100).

Examining every logic path can also be simplified to invoking each point of entry to all routines (although ideally there should only be one for each) and, inside each routine, covering each possible outcome of every decision branch. In figure 2 we have a routine for adjusting bonus points in a game. It takes the input parameters

Just Testing

A complete set of hand-calculated test data for the example illustrated in the flowcharts might look like this:

	INPUT		OUTPUT
LEVEL	HITS	BONUS	BONUS
6	10	200	1300
4	10	550	2300
7	10	550	3950
4	10	200	800
7	10	200	1400
1	20	2500	2600
1	20	550	550
6	5	200	300
6	50	200	300
4	5	2500	2600
7	50	2500	2600
4	50	550	550
7	5	550	550

BONUS,LEVEL and HITS and returns a (possibly new) value for BONUS.It might be written thus:

```
6030 IF LEVEL>2 AND HITS=10 THEN
   BONUS=BONUS*LEVEL
6040 IF LEVEL=6 OR BONUS>2000 THEN
   BONUS=BONUS+100
```

To cover the outcome of each conditional expression, we need to consider the inputs to each that would cause an output of 'yes' or 'no'. In both decisions we are looking at the effects of two variables combined by a logical operator (AND and OR). This means that we have to take the combined values of the variables and not their individual values into consideration. To see why, consider what would happen if we tested values for LEVEL of 4 and 1 and for HITS of 10, 5 and 20 in the first decision. When LEVEL=4, the three values of HITS are tested but when LEVEL=1 they are not. This is a case of part of a decision 'masking' another part. So that we can test each part separately, it is best to simplify compound decisions.

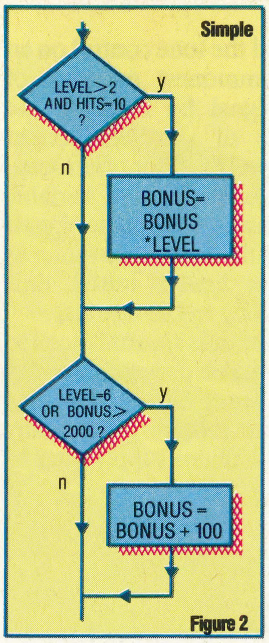


Figure 2

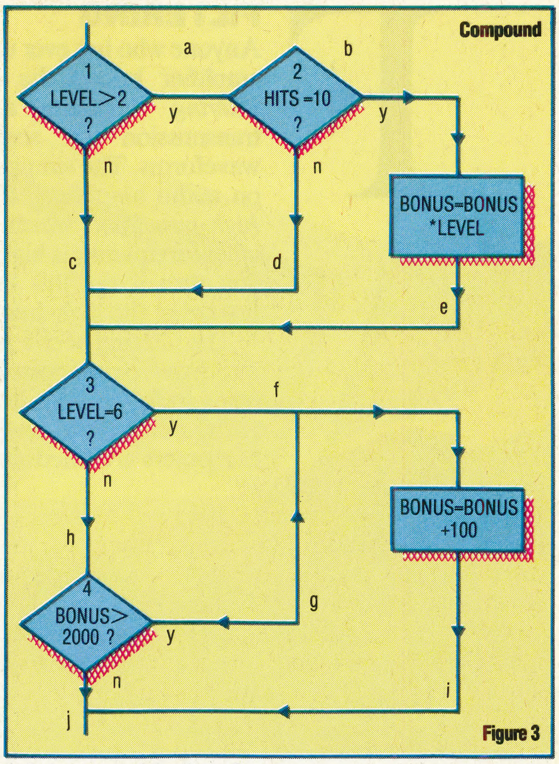


Figure 3

Decision Masking
Simplifying compound decisions and labelling the flowchart links makes systematic testing easier

Looking at figure 3, we can see that with four binary decisions there are 2^4 (=16) possible outcomes and we must cover them all. A start is to list the conditions for a yes or no outcome for each decision like this:

	1	2	3	4
yes	LEVEL>2	HITS=10	LEVEL=6	BONUS>20000
no	LEVEL=2 LEVEL<2	HITS<10 HITS>10	LEVEL<6 LEVEL>6	BONUS=2000 BONUS<2000

These can then be used to derive the values for representative test data. For instance, for the path taking the route adfi (see figure 3), LEVEL must be greater than 2 and equal to 6, HITS must be not equal to 10 and BONUS may be any value (because it is not involved). The values LEVEL=6, HITS=20 and BONUS=150 would exercise this path — as would many others, of course. The route abehj could be tested with LEVEL=4, HITS=10 and BONUS=600 (don't forget we are talking about the *input* value of BONUS that may later be multiplied with LEVEL).

Equally importantly, the results that should be produced by each set of test data should be calculated before the test run so that the results can be compared. The input data on their own will merely test whether the program runs. To test that it is doing what it should, the output must be calculated (by hand) beforehand. A complete set of test cases for this example is shown (left).

Equipped with a method of 'exercising' our software, we now need a way of tackling a large program so that the complexity does not become overwhelming. It is here that another benefit of structured programming is felt. Programs written as a collection of independent modules arranged in a hierarchy allow us to test each module individually. Because the modules are arranged in this way, we can start with the topmost module and work down, testing each individual module only when all of those above it have been tested, and we can use already-tested modules to provide data for those lower in the structure.

The module being tested will have above it (unless it is the first one), a fully tested *driver* module. The modules below it, known as *stubs*, are, so far, untested and therefore unreliable, so they are simulated by short pieces of code that simply return the appropriate test data when called by the module being tested. This arrangement is sometimes known as a *test harness* and it is a framework into which module routines can be put for testing. Figure 4 shows the principle. Modules 1, 2 and 3 have already been tested while modules 5, 6 and 7 are simulated.

One final point must be stressed. Testing is an important part of the program's life cycle and, as such, deserves to be well documented. It pays to keep records of the test data derived for a routine so that, if it shows a bug later, the same tests will not have to be repeated, or the testing can be examined for where it was inadequate.

Top-Down Testing
Testing is made much simpler by the top-down approach, since each module can be tested as it is written, both in isolation and in association with other tested modules. The behaviour of unwritten modules can be simulated by writing 'stubs' — code that artifically generates examples of the module's predicted output

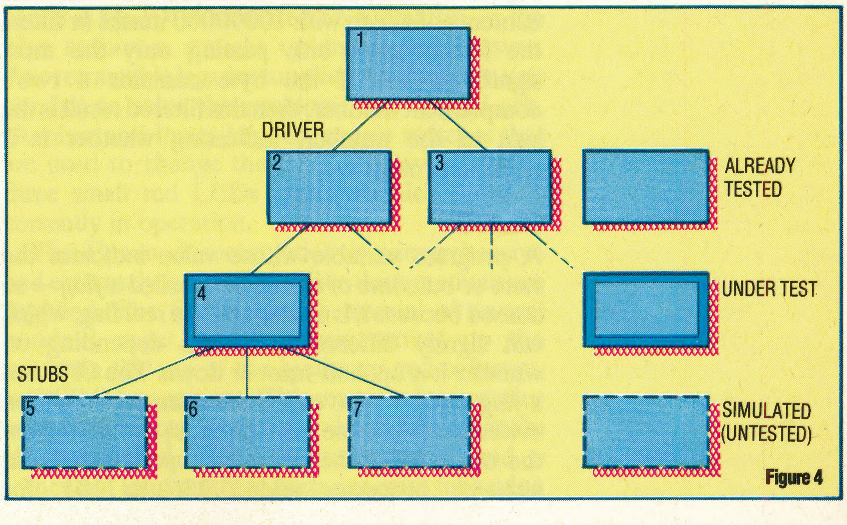


Figure 4

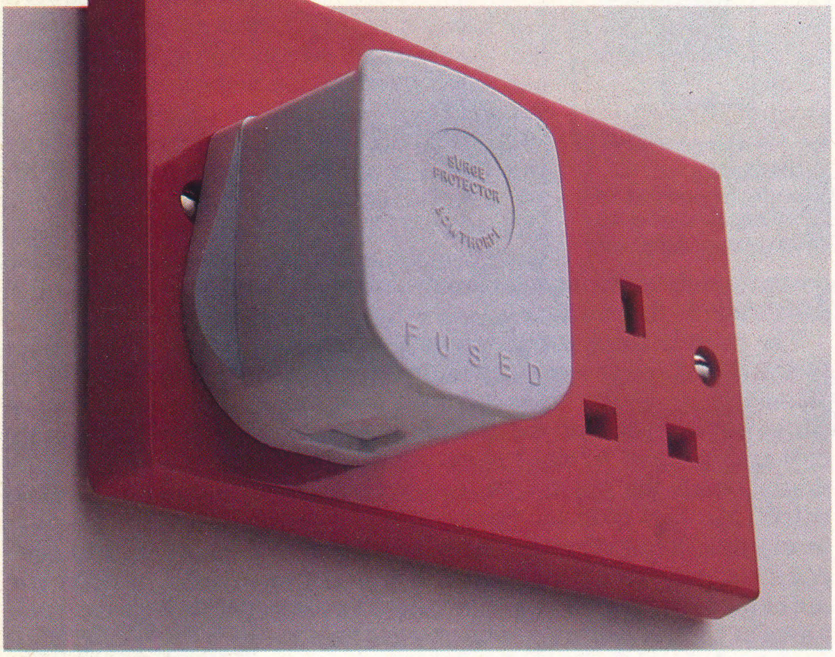
LIZ DIXON

F

FILTERING

Anyone who has ever used the tone control on an amplifier knows the commonest meaning of *filtering* — altering a signal by blocking the transmission of some of its component waveforms. The simple rumble filter often seen on audio amplifiers, for example, is a variable *high-pass* filter, which passes only those signals whose frequency is higher than some set value: as you turn it up, this lower limit is raised, thus favouring the high-frequency treble sounds.

The filtering of computer data transmissions to remove electrical noise is vital in preventing errors being induced; both frequency filters and error/parity coding are used for this purpose. Many computers are fitted with a mains supply filter to block high-voltage 'spikes' or 'transients' (usually induced by the switching of high-current devices such as lift motors or commercial freezers). These pulses might otherwise get past the computer's voltage regulator, and damage the chips or reset the system.



IAN McKINNELL

Plug-In Protection
Filtering the mains power supply using a plug-in unit like this to suppress high voltage 'spikes' or current surges can save computer users hours of wasted effort by eliminating the accidental resets that may be caused by such transients

Information can be filtered, or *masked*, by logical operations. For example, ANDing the contents of a byte with 10000000 masks or filters the lower seven bits, passing only the most significant bit. If the byte contains a two's complement number, then this filtered result is the sign of the number, indicating whether it is positive or negative.

FLAG

A program variable whose value indicates the state or outcome of a process is called a *flag* — so named because it is analogous to a real flag, which can signify different meanings depending on whether it is up, half-mast or down. The CPU has a *flag register* (also called the *status* or *condition code register*) whose bits (the flags) are set to show the outcome of the processor operations. If an eight-bit processor added 236 to 101, for example, then the zero flag would be cleared, showing a non-zero result, and the carry flag would be set, showing that the result was greater than 255 — the accumulator's numerical limit. Some of the processor's Assembly language instructions vary their actions according to the state of these flags, thus allowing program decision-making.

FLIP-FLOP

Since computers run on binary logic, an electronic device that exhibits two stable states, and will switch states in a predictable way, is an essential requirement for constructing either memory or logic circuits. This essential component is called a *flip-flop*, or bistable (see page 168), and it is usually made by cross-connecting two NAND gates so that the output of each is an input of the other (see page 228).

FLOATING POINT NOTATION

A computer usually has at least two formats for its internal representation of numbers: *integer* and *floating point* notation. Integer notation is reasonably straightforward — two bytes are allocated to each integer variable (e.g. X%) in signed 16-bit notation; the range of possible integers, therefore, is:

Decimal	−32768		+32767
Binary	10000000 00000000	to	01111111 11111111

In floating point notation, however, numbers are represented thus:

$$+\text{number} = +\text{mantissa} \times \text{base}^{+\text{exponent}}$$

$$\text{e.g.} + 317440 = +.60546875 \times 2^{+19}$$

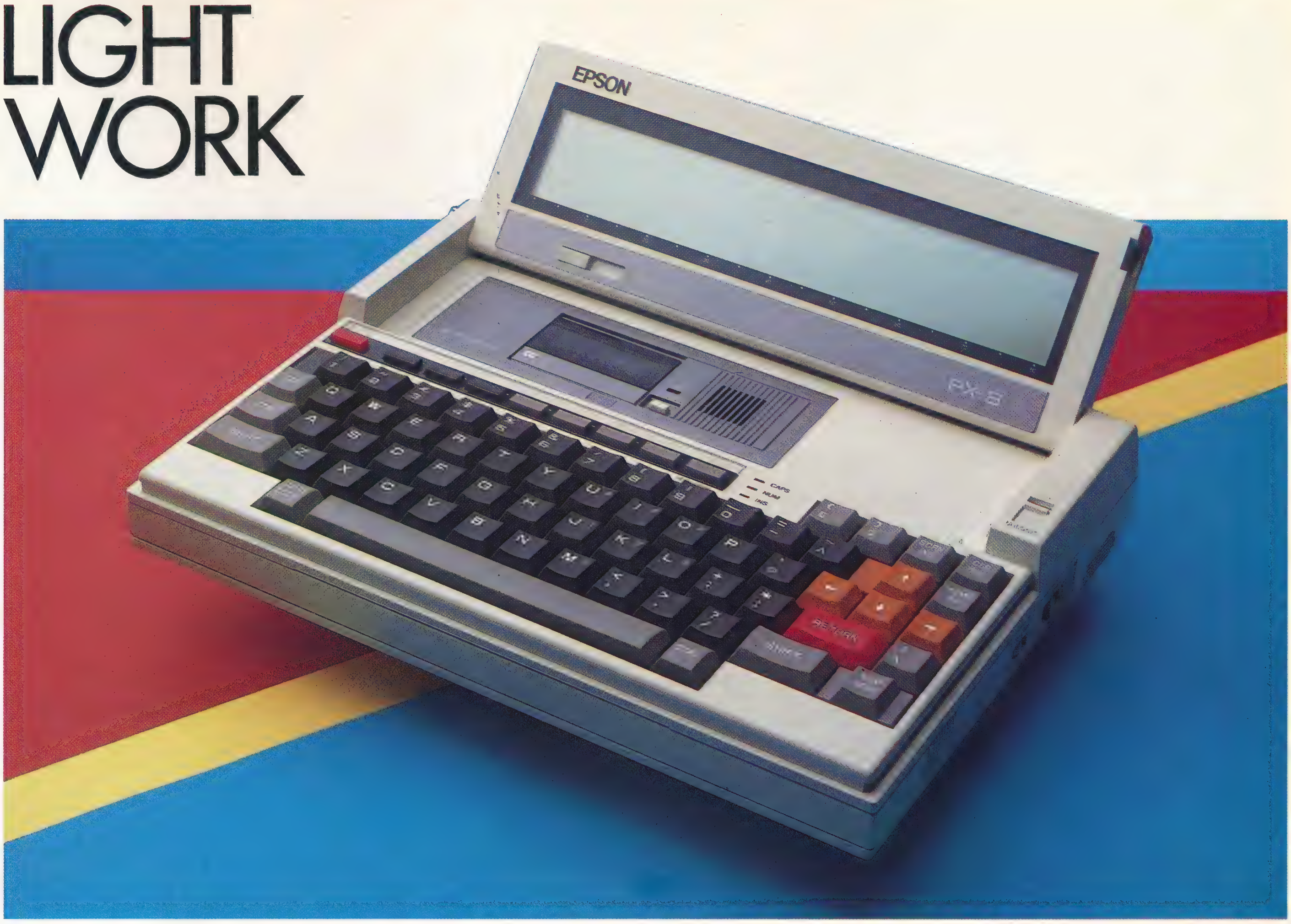
Since, in a binary machine, the internal number base is always two, there is no need to store it, so only the signed mantissa and exponents are saved, thus:

Exponent		Mantissa			
S		S			
0	0010011	0	1001101	10000000	00000000
byte 0		byte 1		byte 2	byte 3

Both the mantissa and exponent are stored in *two's complement* form, so that the most significant bit of each is the sign bit. If the mantissa is adjusted so that it is always in the range (0.5 ⩽ mantissa < 1.0 — or, in binary fractions, 0.1 ⩽ mantissa < 1.0) then it is said to be *normalised;* this means that the first bit of the mantissa after the sign bit will always be one.

The great advantage of floating point notation is that it allow compact storage of very large and very small numbers. As more bits are allocated to the mantissa, so the precision of this format increases; adding bits to the exponent extends the range of expressible numbers. Furthermore, the exponential format makes writing efficient arithmetic routines — especially for multiplication and division — reasonably easy.

LIGHT WORK



Latest Offer
The PX-8 lap-held computer is made by Epson, famous for its dot matrix printers, the HX-20 portable, and the QX-10 desktop business computer. The PX-8 comes with 64K of RAM, a Liquid Crystal Display screen, CP/M, and several software packages for £798

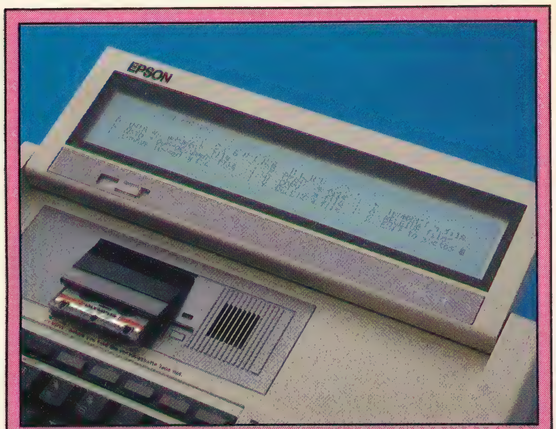
Portable computers can be fairly large, 'luggable' machines or they may be small enough to fit into a pocket. Between these two extremes are the 'lap-helds'. One of the latest contenders in this sector of the market is Epson's PX-8, a portable machine with 64 Kbytes of RAM, CP/M and a collection of software supplied.

The PX-8 comes in a case the size of a telephone directory, and weighs approximately 2.3 kgs (5 lbs). The casing is finished in two tones of beige, with a sliding metal handle, and at first sight the package looks very little like a computer. However, part of the case slides off to reveal a full-featured computer keyboard and a folded-down display screen. The screen is released by moving a sliding switch marked 'UNLOCK', which also reveals a microcassette tape recorder. The screen panel is ratchet-controlled and may be placed in any one of 11 positions, although only five or six of these give a good viewing angle.

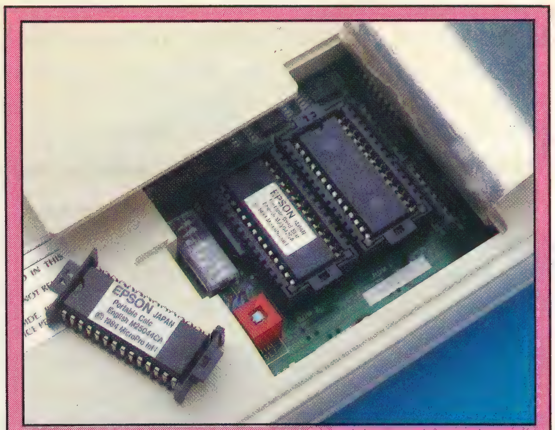
The keyboard has 72 typewriter-style keys, colour-coded to indicate their usage. The dark brown alphanumeric keys are arranged in standard QWERTY format on the US and UK versions (there is also a French AZERTY model for sale in Europe), with the '£' sign on the English keyboard replacing the '#' (hash mark) on the American version. (All the 'international' characters may in fact be accessed from any of the keyboards by changing the PX-8's DIP switches. This process is explained clearly in the user's manual.) There are also four bright orange cursor control keys, Insert, Delete and Home keys, three system function keys (Escape, Pause and Help), and five programmable function keys.

The keyboard is manufactured to a high standard and is especially easy to use if the machine is lap-held. However, the keyboard is less useful if the PX-8 is located on a desktop, as the keys require a straight up-and-down pressure. Two retractable legs are supplied; these tilt the unit but fail to solve the problem. The Caps Lock, Number and Insert keys are toggle switches that are used to change the PX-8 display modes — three small red LEDs indicate which mode is currently in operation.

The Epson documentation is comprehensive and extremely well written. Two thick manuals are supplied. The first is a user's manual of several hundred pages, which covers setting up the machine, the use of the hardware and software, and CP/M operations. This manual also includes memory maps, complete lists of available characters and their associated codes, and a somewhat long machine code program for saving



Big Screen
The PX-8 has an 8-line by 80-character LCD screen that runs off the machine's battery. The screen provides 480 by 64 pixel resolution for graphics displays



ROM Exchange
Pulling back a small panel on the underside of the PX-8 reveals the slots that hold ROM-based software. Portable Wordstar is installed in the machine, along with the CP/M operating system. To switch from Wordstar to Calc, you simply exchange the ROM chips

and loading the graphics screen from disk. The second volume is a superb BASIC programming reference guide, which is also several hundred pages long. This book begins by explaining how to instal and use BASIC (supplied, like the bundled software, on a ROM 'capsule'), before going on to a clear discussion of the nature of programming, an examination of the various PX-8 display modes, and a detailed breakdown of all BASIC commands available.

The PX-8 uses a Z80-compatible CMOS CPU. CMOS (Complementary Metal Oxide Semiconductor) chips require considerably less power than standard CPU chips, and this fact, together with the PX-8's use of a low-power LCD screen, enables the unit to be run entirely on battery power. Two battery units are supplied — one for main power use and the other as a back-up. The battery must first be charged before the computer can be used, so an eight-hour wait must be expected between first setting up the machine and actually using it. The main unit is rechargeable and gives up to 15 hours of continuous operation before charging is necessary. Epson claims a life expectancy of three to four years for this unit.

Once the PX-8 is ready to go, the operating system must be initialised. The steps needed to accomplish this are explained in detail in the manual; these involve entering the day, date and time, and taking care of a few 'housekeeping' tasks. One of these is the formatting of a RAM disk. The PX-8 has the ability to set aside a portion of RAM — user selectable between nine Kbytes (the default value) and 24 Kbytes — for use as a 'disk' storage device. The operating system treats this area of memory in exactly the same fashion as it would an external disk drive. Before use, the RAM disk must be formatted and the amount of RAM to be used specified. Epson also provides an add-on RAM disk unit, containing 120 Kbytes of extra RAM at a price of £270.

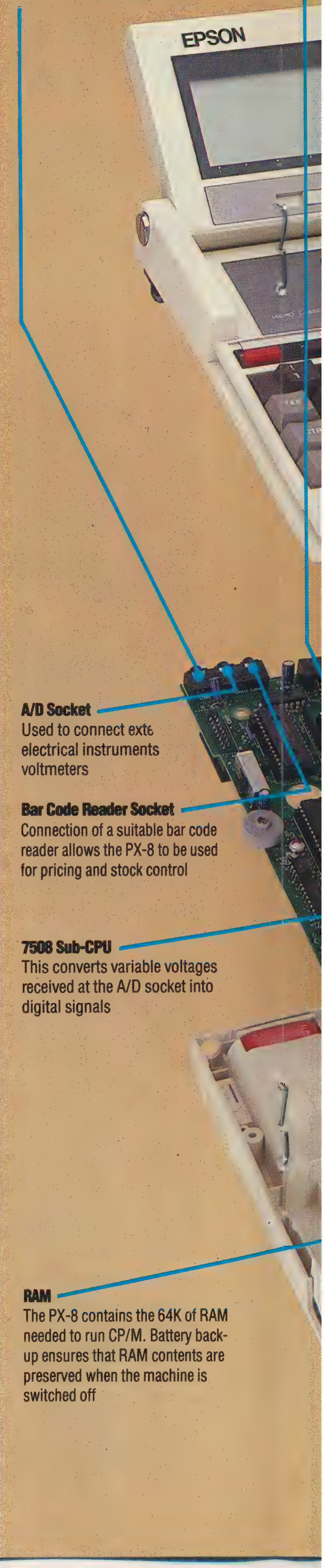
Once these details have been taken care of, the PX-8 loads the CP/M operating system from ROM and displays a CP/M utilities and ROM software directory in menu form on the LCD screen. Software in any of three formats may be used — cassette, disk or ROM. ROM software is held on EPROM chips that slide into a socket located underneath the machine. The software supplied with the PX-8 — Portable Wordstar, Portable Calc and Portable Scheduler — is supplied in this 'capsule' format, as is the BASIC interpreter. To select a particular application, the cursor keys are used to indicate the desired choice and Return is then pressed. The chosen program is loaded from ROM (addressed by the PX-8 as drives A and B) into RAM (addressed as drive A).

The LCD screen gives an eight line by 80-column display, with a graphics resolution of 480 by 64 pixels. The greatest drawback of this type of screen — and in fact the only major disadvantage of this excellent machine — is the slowness of the display. Characters appear quickly enough as they are typed in, but any erasures — especially those involving whole words or sentences — are slow.

The software supplied with the PX-8 is fairly comprehensive. In addition to the word processor, spreadsheet and database already mentioned, Epson provides a telecommunications program for use with a modem, and a program that allows files to be transferred from the PX-8 to larger machines, such as Epson's QX10. And, as the PX-8 is a CP/M machine, much existing CP/M software should also be usable.

PX-8 BASIC is Epson-enhanced Microsoft, including AUTO line numbering and renumbering, a full screen editor, graphics and sound commands, statements that support communications through the built-in RS232 interface, and commands that enable the microcassette recorder to be used as if it were a disk drive (for direct access storage).

All in all, the Epson PX-8 is a marvellous computer. It is ideal for business executives, for journalists, or for anyone who needs a small powerful computer that may be used on the move. With its excellent features and reasonable price, the PX-8 is in a class of its own among portables.



Speaker Out
Provides the facility to boost sound output by connecting the PX-8 to an external speaker

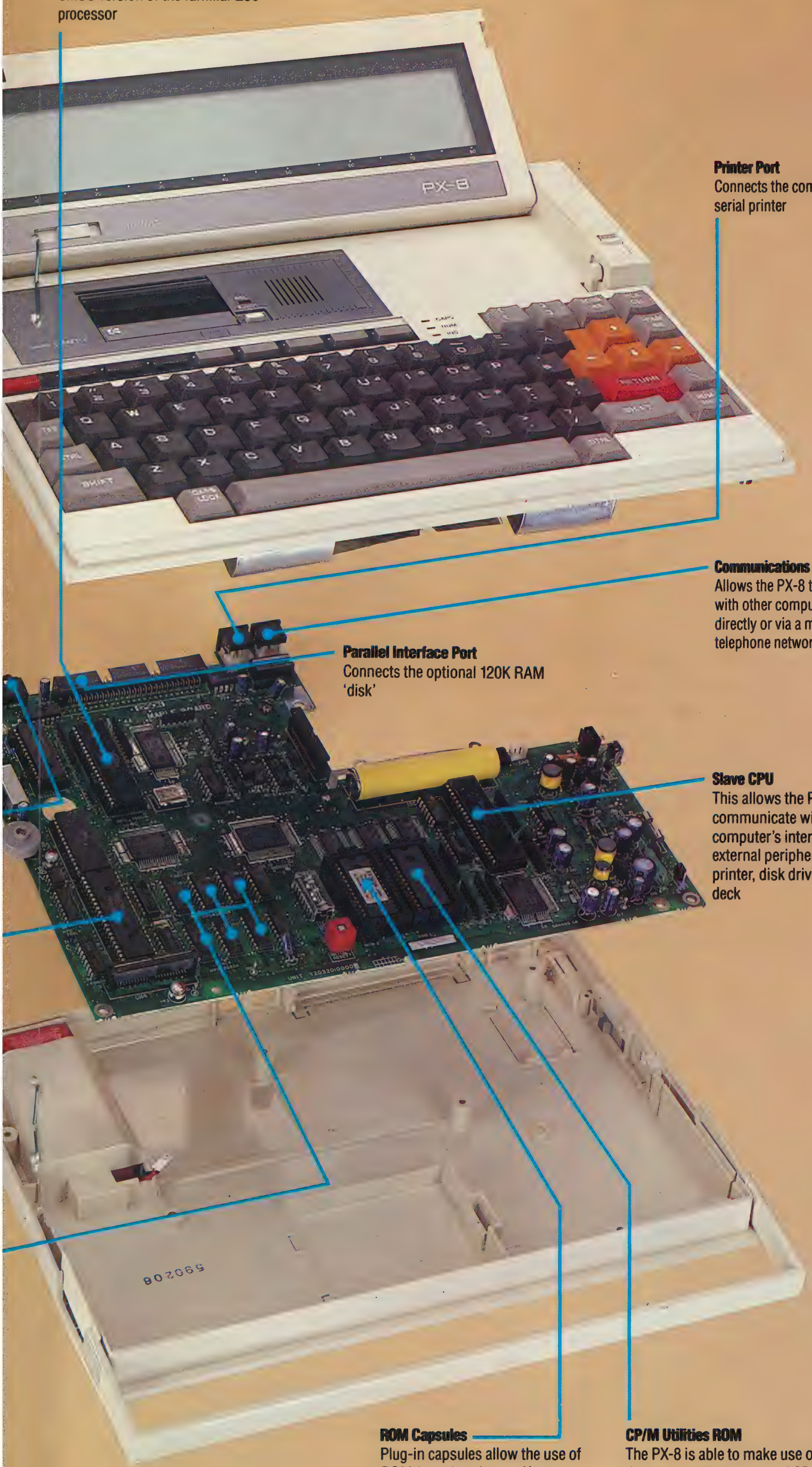
CP
CM
pro

A/D Socket
Used to connect exte
electrical instruments
voltmeters

Bar Code Reader Socket
Connection of a suitable bar code reader allows the PX-8 to be used for pricing and stock control

7508 Sub-CPU
This converts variable voltages received at the A/D socket into digital signals

RAM
The PX-8 contains the 64K of RAM needed to run CP/M. Battery back-up ensures that RAM contents are preserved when the machine is switched off



CPU
CMOS version of the familiar Z80 processor

Printer Port
Connects the computer to external serial printer

Communications Socket
Allows the PX-8 to communicate with other computers, either directly or via a modem and telephone network

Parallel Interface Port
Connects the optional 120K RAM 'disk'

Slave CPU
This allows the PX-8 to communicate with the computer's internal speaker and external peripherals such as a printer, disk drive or cassette deck

ROM Capsules
Plug-in capsules allow the use of ROM-based packages. Here Wordstar is fitted, but this may be replaced by another ROM capsule, such as Portable Calc

CP/M Utilities ROM
The PX-8 is able to make use of the large available range of CP/M software

CHRIS STEVENS

EPSON PX-8

PRICE
£798

DIMENSIONS
297 x 216 x 48mm

CPU
Z80-compatible CMOS CPU, 2.4 MHz

MEMORY
64K RAM, 32K ROM plus 6K video RAM

SCREEN
Text: 80 columns x 8 rows. Graphics: 480 x 64 pixels

INTERFACES
RS232C, serial, bar code reader, analogue input

LANGUAGES AVAILABLE
Enhanced Microsoft BASIC operating under CP/M

KEYBOARD
72 key typewriter-style, QWERTY format, including cursor control and five programmable function keys. 12 keys can be used to form a numeric keypad

DOCUMENTATION
Two large ring-bound volumes, an operations manual and BASIC reference guide. Both are very thorough, and well written

STRENGTHS
Wide LCD screen (80 characters) makes it easy to follow what is happening on the screen; ROM-based software simplifies loading programs into memory; easily expandable

WEAKNESSES
LCD screen shows only eight lines and is slow to manipulate. Even with excellent documentation, CP/M is not a very friendly operating system, especially for a first-time user

DUAL CONTROL

In the last section of this series we wrote the software to control a Lego car with one motor. We were able to move the car in two directions by simply switching the current from the output box. Now we extend these principles to control a Lego car powered by two motors.

If we use two motors of equal power to drive a vehicle, we can gain computer control over all directions of movement by combining the forward and reverse movements of each motor. This allows us to turn the vehicle as well as direct it forwards or backwards. There are, in fact, two methods of turning a twin motor vehicle; the first of these is simply to stop one motor while turning the other. This will cause the vehicle to turn in an arc, pivoting about the stationary wheel(s). The second method involves turning one motor backwards as the other turns forward, improving the manoeuvrability of the vehicle as, in turning, the vehicle will pivot about its central axis.

We can control each motor bi-directionally by using the four red outputs on the low voltage output box (built on page 574) and connecting the right-hand motor to terminals 0 and 1, and the left-hand motor to terminals 2 and 3 (positive and negative, respectively).

Each motor is connected across a pair of adjacent positive output terminals so that we can have independent directional control of each motor. By placing the appropriate number in the data register we can now make the vehicle move forwards or backwards, or turn to the left or right. The right-hand (RH) motor will go forward if line 0 is set high and line 1 is set low, and go backwards if line 1 is set high and line 0 is set low. Similarly, the left-hand (LH) motor will go forwards if line 2 is high and line 3 is low, and backwards if the converse is true. By combining these movements we can control the motion of the whole vehicle:

Vehicle Motion	LH Motor	RH Motor	Bit Pattern	No In DATREG
OFF	OFF	OFF	0000	0
FORWARD	FORWARD	FORWARD	0101	5
REVERSE	REVERSE	REVERSE	1010	10
PIVOT LEFT	FORWARD	REVERSE	0110	6
PIVOT RIGHT	REVERSE	FORWARD	1001	9
ARC LEFT	FORWARD	OFF	0100	4
ARC RIGHT	OFF	FORWARD	0001	1

The following program allows us to control the vehicle directly from the keyboard using 'T' for forwards, 'B' for reverse, 'F' to turn left, and 'H' to turn right. If no key is pressed then the vehicle will stop.

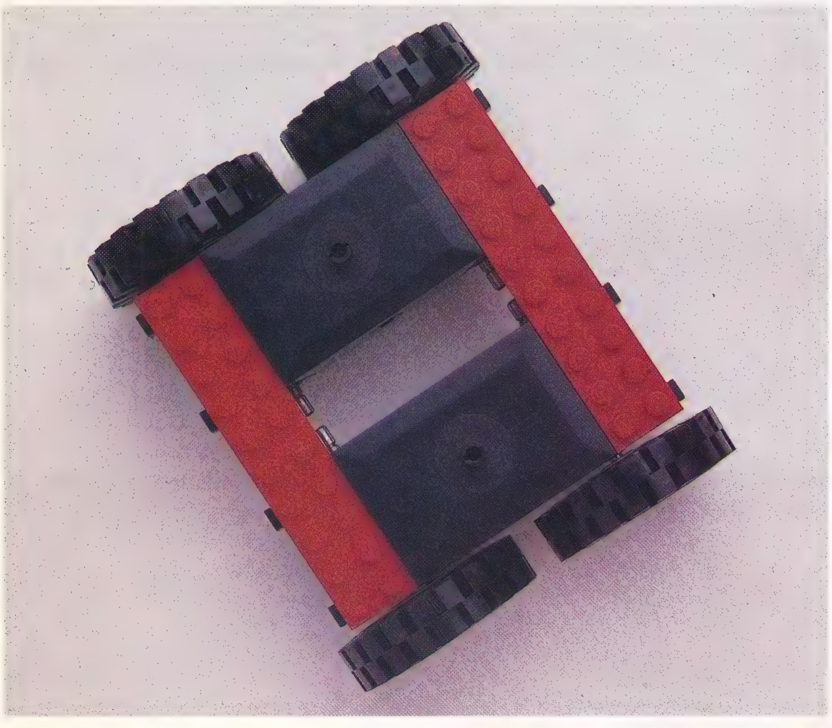
BBC MICRO

```
  10 REM BBC TWIN MOTORS
  20 DDR=&FE62:DATREG=&FE60
  30 ?DDR=255
  40 REPEAT
  50 A$=INKEY$(10)
  60 PROCtest_keyboard
  70 UNTIL A$="X"
  80 ?DATREG=0
  90 END
1000 DEF PROCtest_keyboard
1010 IF A$=" " THEN ?DATREG=0
1020 IF INKEY(-36) = -1 THEN ?DATREG=5
1030 IF INKEY(-101) = -1 THEN ?DATREG=10
1040 IF INKEY (-68) = -1 THEN ?DATREG=6
1050 IF INKEY (-85) = -1 THEN ?DATREG=9
1060 ENDPROC
```

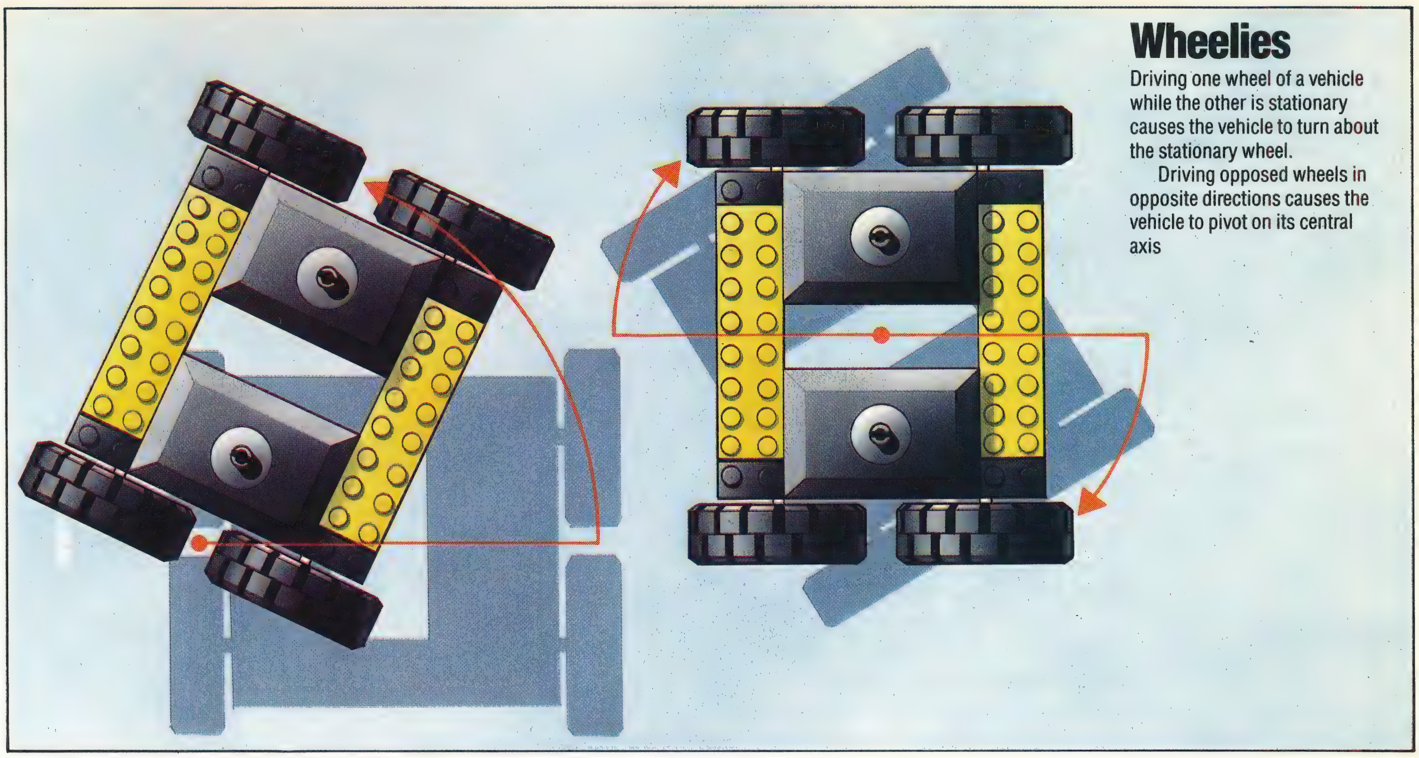
COMMODORE 64

```
  10 REM CBM 64 TWIN MOTORS
  20 DDR=56579:DATREG=56577
  25 POKE650,128: REM REPEAT KEY MODE
  30 POKE DDR,255
  40 GETA$
  50 GOSUB1000:GOTO70
  60 POKEDATREG,0
  70 IF A$<>"X" THEN FOR I=1TO100:NEXT:GOTO40
  80 POKE DATREG,0
  90 END
1000 REM TEST INPUT S/R
1005 IFA$=" " THEN POKE DATREG,0
1010 IFA$="T" THEN POKE DATREG,5
1020 IFA$="B" THEN POKE DATREG,10
1030 IFA$="F" THEN POKE DATREG,6
1040 IFA$="H" THEN POKE DATREG,9
1050 RETURN
```

Tandem Turtle
Having discovered how to drive the Lego vehicle backwards and forwards, we can now link two of them to make a buggy. The motors can be switched individually, making the new vehicle far more manoeuvrable than the old



IAN McKINNELL



Wheelies

Driving one wheel of a vehicle while the other is stationary causes the vehicle to turn about the stationary wheel.

Driving opposed wheels in opposite directions causes the vehicle to pivot on its central axis

KEVIN JONES

In each version of the program the vehicle will move only while a key is being depressed. As soon as the key is released, the motors are turned off by placing a zero in the data register. The program is exited in each case by pressing the 'X' key.

In the BBC version of the program the procedure TEST-KEYBOARD allows us to test the keyboard directly, rather than reading the keyboard buffer, by using INKEY. This allows more responsive control of the vehicle. The Commodore 64 version firstly turns on the keyboard auto-repeat so that if a key is held down it will keep sending characters into the keyboard buffer to be read by the GET command. Unfortunately there is no way of reading the keyboard directly and accurate control is therefore more difficult than on the BBC Micro. Responsiveness can be improved by clearing out the keyboard buffer just prior to reading it. Inserting the following line into the Commodore version of the program will achieve this.

```
35 GET J$:IF J$<>""THEN35
```

In addition, the GOTO at the end of line 60 should be changed to GOTO35.

The speed at which a key repeats when held down can cause a problem with both versions of this program. If the main program loop is executed faster than the key repeat time, then when the routine comes to test for a keypress again it will think that no key is being pressed. This will result in a rapid switching on and off of the motor as the output alternates rapidly between that set for the chosen direction and zero. In each of the versions of the program this problem has been obviated by adding code to slow down the execution time of the main program loop. In the BBC version using INKEY$(10) causes the computer to 'hang around' for 10 hundredths of a second, waiting for an input, before moving on. In the Commodore 64 version a short delay loop has been added in line 60. The values of these delays were found by a process of trial and error, and are dependent on the length of time needed to execute one pass of the routine. You may find, when writing your own programs, that the routine execution time exceeds the key repeat speed; if not then simply insert a short delay into your code.

Now that we have gained control over the movements of our vehicle it is interesting to design a program that will 'memorise' a sequence of moves and replay them. To do this we can make use of a two-dimensional array that records direction and the time taken for each different manoeuvre made. The first part of such a program will be the same as those already given but the second part will replay the stored data. The data will be stored in an array DR(), where DR(C,1) stores direction and DR(C,2) stores the time taken for each movement. A new element in the array is used each time a new direction is selected. This condition is indicated by a change in the contents of the data register. A counter, C, is used to keep track of the array elements.

BBC MICRO

```
1000 REM BBC MOVEMENT MEMORY
1010 DDR=&FE62:DATREG=&FE60
1020 DIM DR(100,2)
1030 ?DDR=255:C=1:REM INIT COUNT
1040 REPEAT
1050 A$=INKEY$(10)
1060 PROCtest_keyboard
1070 UNTILA$="X"
1080 ?DATREG=0
1090 DR(C-1,2)=TIME
1100 REPEAT A$=GET$
1110 UNTIL A$="C"
```

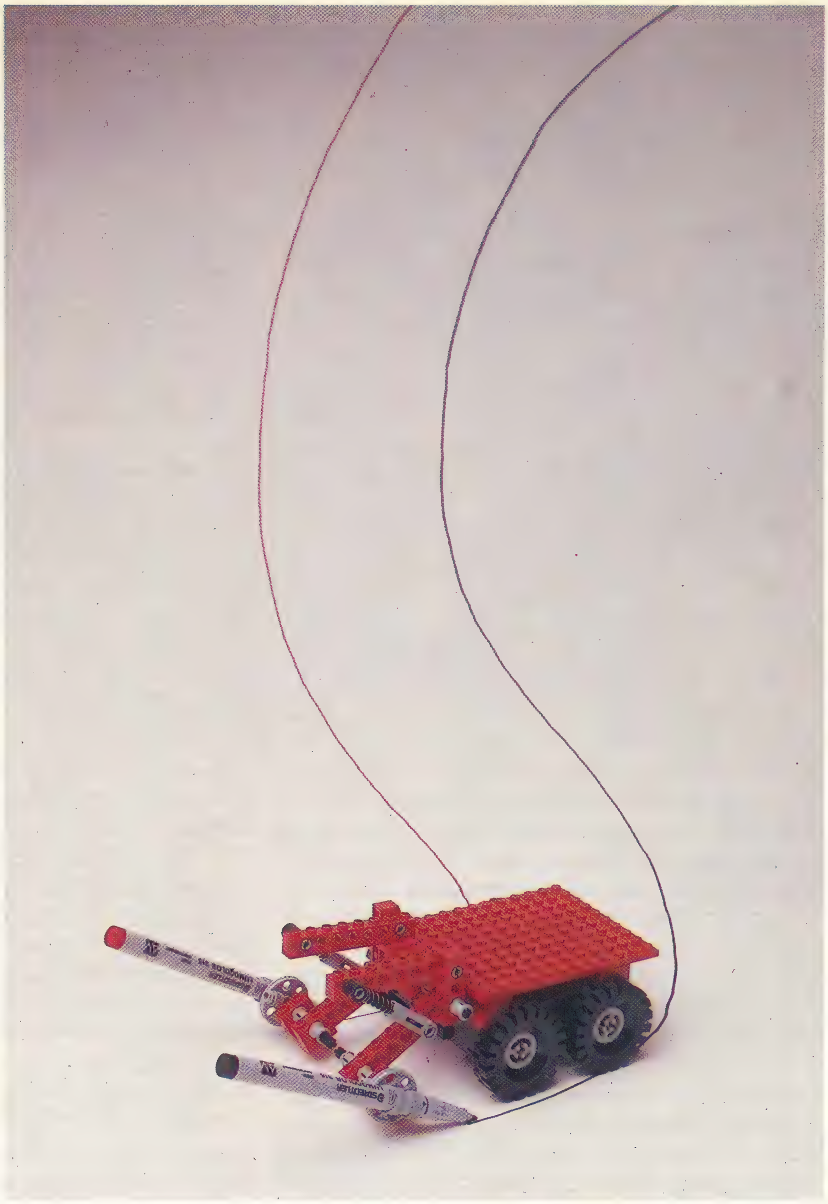
```
1120 REM REPLAY DATA
1130 FOR I=1TOC
1140 ?DATREG=DR(I,1)
1150 TIME=0
1160 REPEAT UNTIL TIME>=DR(I,2)
1170 NEXT I
1180 END
1190 :
1200 DEF PROCtest_keyboard
1210 IFA$=" " THEN ?DATREG=0
1220 IF INKEY(-36) = -1 THEN ?DATREG=5
1230 IF INKEY(-101) = -1 THEN ?DATREG=10
1240 IF INKEY(-68) = -1 THEN ?DATREG=6
1250 IF INKEY(-85) = -1 THEN ?DATREG=9
1260 PT=?DATREG
1270 IF PT<>DR(C-1,1) THEN PROCadd_data
1280 ENDPROC
1290 :
1300 DEF PROCadd_data
1310 DR(C-1,2)=TIME: REM STORE LAST TIME
1320 TIME=0: REM START NEW TIME
1330 DR(C,1)=PT: REM STORE PORT STATUS
1340 C=C+1: REM INCREMENT COUNT
1350 ENDPROC
```

COMMODORE 64

```
  10 REM CBM 64 MOVEMENT MEMORY
  15 DIMDR(100,2): REM DIRECTION ARRAY
  20 DDR=56579:DATREG=56577
  25 POKE650,128 : REM SET KEY REPEAT MODE
  30 POKEDDR,255: REM ALL OUTPUT
  35 C=1: REM INITIALISE COUNT
  40 GETA$
  50 GOSUB1000: REM TEST INPUT
  70 IF A$<>"X" THEN FOR I=1TO200:NEXT:GOTO40
  80 POKE DATREG,0: REM OFF
  85 DR(C-1,2)=TI-T: REM ENTER LAST TIME
  90 STOP: REM TYPE 'CONT' TO CONTINUE
  95 REM REPLAY DATA
 100 FOR I=1TOC
 110 POKEDATREG,DR(I,1)
 120 T=TI
 130 IF (TI-T)<DR(I,2)THEN130
 140 NEXT
 150 END
 999 :
1000 REM TEST INPUT S/R
1005 IFA$=" " THEN POKEDATREG,0
1010 IFA$="T" THEN POKEDATREG,5
1020 IFA$="B" THEN POKEDATREG,10
1030 IFA$="F" THEN POKEDATREG,6
1040 IFA$="H" THEN POKEDATREG,9
1045 PT=PEEK(DATREG)
1050 IFPT<>DR(C-1,1)THENGOSUB1500
1498 RETURN
1499 :
1500 REM ADD DATA TO ARRAY
1510 DR(C-1,2)=TI-T: REM ADD LAST TIME
1520 T=TI: REM TAKE NEW TIME
1530 DR(C,1)=PT: REM ENTER CURRENT PORT CONTENTS
1540 C=C+1: REM INCREMENT COUNT
1999 RETURN
```

This program allows the user to move the vehicle about under keyboard control. As each move is recorded as a direction and a time interval, any errors introduced in the timing of each movement will produce errors in the replay. We are entering into the difficult area of real-time computing, where program structure and execution time can become important factors.

In the next instalment of Workshop we shall take control one stage further by bringing our twin motor vehicle under the control of a joystick.



Memory Movements
It is reasonably simple to write a buggy-controlling program that accepts directions from the keyboard and drives the car accordingly. It is not much more difficult to extend the program so that it stores the operator's commands, and then replays them to the buggy, thus reproducing — in theory — the previous pattern of movement. Comparing the original with the supposed duplicate gives a measure of the software problems caused by dealing with the real world: the computer works in exact numbers and times on a simplistic model of a perfect universe, not allowing for inertia, frictional losses, irregular surfaces and low-tolerance engineering. In the light of this experience, the performance of LOGO-driven floor turtles is impressive

Exercises

Now that we can control a vehicle's movement in all directions, many possibilities arise for short programming exercises. You can probably think of many but here are a few ideas:

1) Try to calibrate your vehicle. How long does the relevant number have to be in the data register to make the vehicle move one metre forwards or backwards, or turn through 90 degrees?

2) Design an obstacle course for your vehicle and, using the programs given as a basis, write a program that allows you to 'teach' the vehicle to negotiate the course under keyboard control. Once you have guided the vehicle through its course, the program should take over, guiding the vehicle back to its starting point and retracing the course.

3) Connect up four switches to the buffer box that allow you to control the vehicle externally from the user port.

INVASION FORCE

Never has a game captured the public's imagination so completely as Space Invaders. Simple yet effective, it proved so successful in the arcades that versions for the most popular home computers were eventually developed. We look at the original Atarisoft version of the game.

Almost every home computer now marketed has a version of Space Invaders available. The game has become so well known that, like Hoover or Biro, it is now frequently used as a generic term — to the extent that anyone playing any arcade game is often said to be 'playing Space Invaders'.

When Space Invaders was launched in 1978 it quickly produced a fever of almost epidemic proportions. Parents became worried that children would spend all their time and money hanging around in unsavoury arcades. What the watchdogs did not realise was that these children were in fact investigating the future.

It can be said that Space Invaders changed the way society saw computers. Before the game came along to exploit the graphics capabilities of the microprocessor, computers were considered untrustworthy, the classic example being the paranoid 'HAL' featured in *2001.*

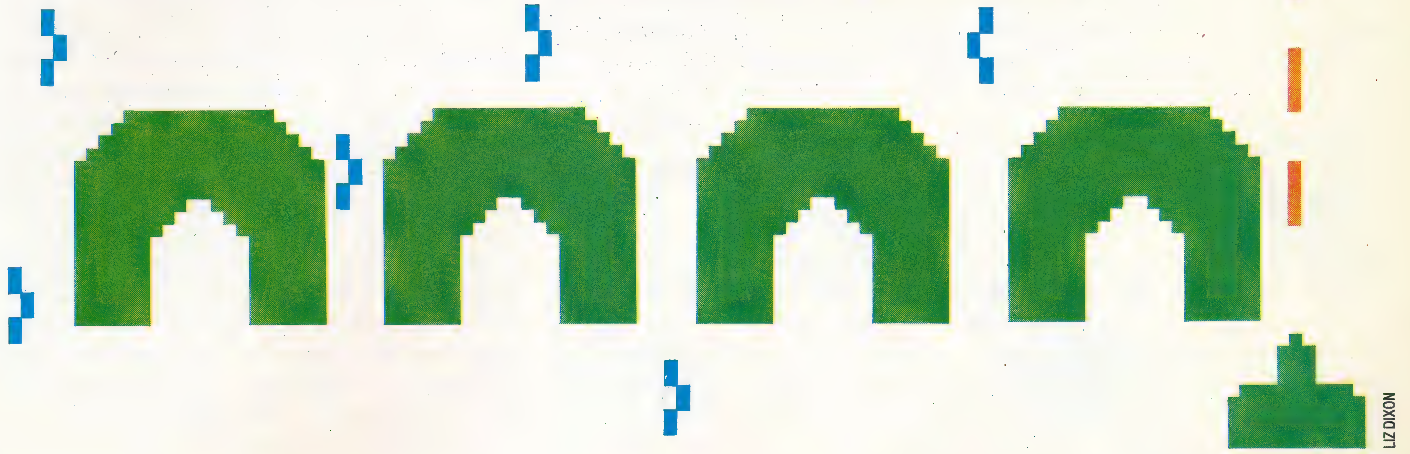
Space Invaders was the forerunner of the whole shoot-em-up computer game genre. Since then there have been literally hundreds of games produced where the one hero or heroine has had to face hordes of attacking nasties, with only their speed on the fire button (and three lives left) to aid them.

It is an undeniable fact that Space Invaders is now showing its age. By today's standards, the game is very simple — yet no other piece of software has captured the public imagination to such a great extent. The player controls a movable laser base, which is used to fire at the massed ranks of invading aliens that move menacingly down the screen towards the Earth's surface. A 'life' is lost if the laser base is hit by alien fire, or if the invaders reach the bottom of the screen.

There are several differences between the arcade original and the versions available for home computers. Instead of appearing out of thin air, the invading hordes now emerge from a large rocket that is situated to the left of the television screen. The invaders themselves are more brightly coloured and the sprites that form them are more complex. The defensive barriers, behind which the laser base could hide in the arcade version, are now missing, and the invaders have a shorter distance to travel before reaching the bottom of the screen. But one factor has remained constant: the menacing 'heartbeat' sound that accompanies the aliens' descent. This becomes more insistent as the invaders get closer and closer, and serves to generate a heady rush of adrenalin, which is probably the major reason for the game's huge success. Another feature of both the arcade and home computer versions is the 'mystery' bonus that is awarded if the player manages to hit one of the flying saucers that occasionally cross the screen from left to right.

Space Invaders has now managed to retain its appeal for six years. Despite the availability of considerably more sophisticated pieces of software, Space Invaders remains an exciting and highly enjoyable game — it is truly a 'software classic'.

Space Invaders: For all Atari computers, £9.99
Publishers: Atari Corporation UK Ltd., Atari House, Railway Terrace, Slough, Berkshire.
Authors: Atari
Joysticks: Required
Format: Cartridge



LIZ DIXON

USEFUL POINTERS

We have examined ways of defining characters on the Commodore 64, BBC Micro and Sinclair Spectrum. Here we discuss possible program improvements, and concentrate on the problem of saving and loading our redefined character set into a specified area of the 64's memory.

Now that our character-defining programs are up and running (see pages 572 and 588), it's worth spending some time comparing the three versions and, perhaps, improving them. The Commodore version was written first, since it is the most difficult of the three to program. This version was then translated, virtually line by line, for the other two machines. Partly because of this translation, and partly because space was limited, the screen formatting is rudimentary, and no use is made of colour, sound or hi-res graphics. Improvements in all these areas can, therefore, obviously be made, but will not be discussed here.

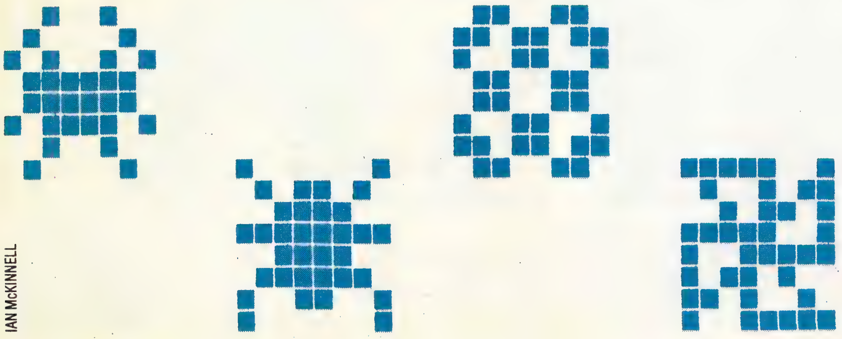
Leaving aside questions of programming efficiency (not really vital in this program, since there are no speed-dependent tasks), we will concentrate on the user interface: instructions, help, command keys and facilities.

There are no instructions in the program, mainly because the listing had to be fitted onto a single page of the course. An instruction page could be printed on the screen at the start of the run, and there is probably room on the main screen display for some abbreviated reminders — a cursor movement display, perhaps, and one-word summaries of the command keys. This should largely remove the need for a help page.

The choice of command keys might be improved. On the BBC Micro and the Spectrum the cursor is moved around the window by the usual cursor control keys, whereas on the Commodore the unshifted function keys are used. This makes for very convenient programming in the Commodore version since the ASCII codes of the eight function keys are consecutive between 133 and 140, but the layout of the keys themselves is not exactly ergonomic and they do not repeat — unlike the BBC and Spectrum keys. This last can be changed on the Commodore by POKE 650,128, but the function keys themselves cannot be made easier to use, so you may wish to restore cursor control to the cursor keys.

Another possible improvement is the choice of cursor movement strategy. As written, this simply disallows as illegal any command that would move the cursor out of the window. The alternative is to 'wrap' the cursor in some fashion: if it is moved off the bottom of the window, it can wrap around to the top of the window, and vice versa, and similarly for horizontal wrapping. This is easily programmed, but requires more code than the simple tests used in subroutine 3500.

The commands provided are the minimum necessary, and could well be extended. On the Spectrum and BBC, SAVEing and LOADing character sets could be included as commands, and in all three versions it would be useful to be able to copy one character's definition to another character — so that CHR$(N) and CHR$(N+1) represented the same character, for example. You might want a hard copy of the new character set, so a printer option could also be added. The program's simple modular structure makes adding these commands reasonably straightforward.



IAN McKINNELL

COMMODORE 64 SAVE

A problem unique to Commodore BASIC is that the SAVE command seems to refer only to the entire BASIC program area, whereas the other two machines' BASICS allow you to specify the area of memory you wish to SAVE. The Commodore LOAD command, however, does allow the loading of files to any desired area, so if we can solve the SAVE problem then we can store and retrieve the new character sets.

The Commodore SAVE command is pointed at the BASIC program area by two address pointers — TXTTAB (at locations 43 and 44), and VARTAB (at 45 and 46). The first, TXTTAB, points to the start of the BASIC program area (usually at address 2048 onwards), while VARTAB points to the start of the BASIC variables area; since this starts where the BASIC program finishes, VARTAB effectively points to the end of the BASIC program area. If we change these pointers so that they indicate the start and finish of the new character set, and then issue a SAVE command, that should solve the problem.

Before we do this, however, we might reconsider the location of the character set itself. The subroutine at line 61000 (see page 573) copies the ROM character set to a two Kbyte block of RAM beginning at 14336, and line 50 sets the top-of-memory pointer below this block, to prevent BASIC from overwriting it. In this way, for the sake of protecting two Kbytes, we are cutting user memory by two-thirds. This presents no problem while running the character generator program, but will be a potential source of difficulty if we load the new character set to that address for use by an applications program that needs more than 12 Kbytes of user memory. Unfortunately, the mechanics of the operating system prevent our locating the character set any higher in memory — if this was possible, we could put it up in the highest two Kbytes of user memory or in the special programs area at 49152 onwards. The solution seems to be to put the character set as low as possible, and move BASIC above it! This can be done by adjusting the contents of the TXTTAB pointers, but it cannot be done from within a BASIC program, and it must be done before the character generator program is loaded into memory.

The sequence of actions, then, is:

1) LOAD and RUN Program 1. This prints the necessary relocation commands onto the screen, so that you can execute them in direct mode, by pressing Return.

2) LOAD the character generator program and make the following changes:

```
61100 CGEN=53248:NCGEN=2048
61500 POKE PO,(PEEK(PO)AND240)OR2
```

and delete line 50.

3) SAVE this new version.

4) LOAD and RUN the character generator exactly as before.

5) When you finish with the program, LOAD and RUN Program 2. Like Program 1, this prints commands on the screen for you to execute.

6) The TXTTAB and VARTAB pointers were set by Program 2, so SAVE "filename" saves the entire two Kbyte character set area between 2048 and 4097.

In future, to run the character generator program you must repeat this sequence, except for step 2.

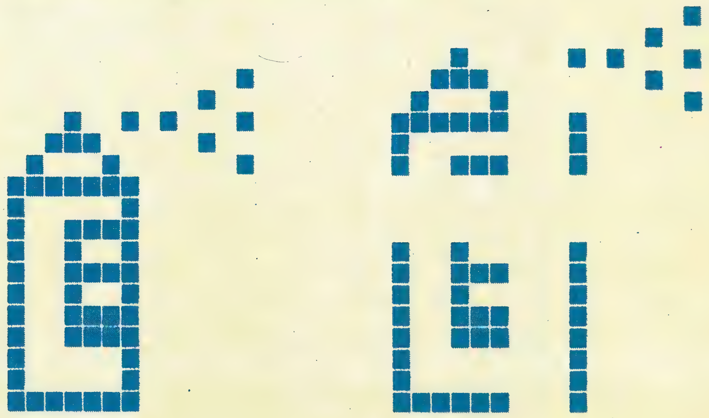
When you want to retrieve the character set, you must LOAD and RUN Program 1, to move BASIC up in memory, and then LOAD the character set thus:

```
LOAD"filename",DN,1
```

where DN (device number) is equal to one for cassette use and eight for the disk drive. The ',1' on the end of the command is known as the secondary address, and is the Commodore way of sending command parameters to peripheral devices. Here, it means that the file is to be loaded to the place in memory from which it was saved, rather than being directed by the TXTTAB pointer into the current BASIC program area. This is possible because when a file is SAVEd the operating system SAVEs the RAM start address as the first data item in the file. When you use the unadorned LOAD command, the start address in the file is ignored in favour of the address pointed to by TXTTAB.

value of X =	bits 3,2,1,0	location pointed to
0	0000	0
2	0010	2048
4	0100	4096
6	0110	6144
8	1000	8192
10	1010	10240
12	1110	12288
14	1110	14336

Value Chart
The command POKE 53272,(PEEK(53272)AND 240)OR X forces the screen display chip to look at the area of RAM containing the redefined character set. The table show the values of X that set the start address of the RAM block.



Once you've re-located BASIC and loaded the new character set, you must make the operating system point at the new character set; this is explained in the table, and is demonstrated in the new version of line 61500 given in this article.

```
199 REM ************************
200 REM*            PROGRAM 1              *
201 REM*        RUN THIS PROGRAM           *
202 REM*      THEN HIT RETURN TWICE        *
203 REM*    THIS MOVES BASIC TO 4096       *
204 REM ************************
300 PRINT CHR$(147):PRINT:PRINT
400 PRINT"POKE43,0:POKE44,16:POKE45,3:POKE46,16"
500 PRINT"POKE4096,0:POKE4097,0:POKE4098,
0:CLR:NEW"
600 PRINT CHR$(19)
```

```
199 REM ************************
200 REM*            PROGRAM 2              *
201 REM*        RUN THIS PROGRAM           *
202 REM*      THEN HIT RETURN TWICE        *
203 REM*     THIS RESETS BASIC PTRS        *
204 REM ************************
300 PRINT CHR$(147):PRINT:PRINT
400 PRINT"POKE43,0:POKE44,8:POKE45,1:POKE46,16"
500 PRINT"POKE4096,0:POKE4097,0:POKE4098,0:CLR"
600 PRINT CHR$(19)
```

MATCH-MAKING

We have already considered the use of indexed addressing on the 6809 processor. Here we examine how this is used to perform simple arithmetic on values in the index registers and discuss the use of subroutines in a string-matching program.

In the previous instalment of the course we took our first look at indexed addressing on the 6809 processor. In indexed mode addressing, the effective address specified by, for example, OFFSET,X is formed as the sum of the offset (which can be a constant or the contents of a memory location) and the current value held in the index register specified (in this case, the X register). We saw that in some common situations the offset may be zero, in which case we can write ,X (although 0,X would also work). In special cases, one of the accumulators A, B or D can be used for the offset (e.g. B,X). And we took a look at how one of the most common uses of indexing — stepping through a table of values — can be made easier by the use of auto-increment and auto-decrement mode. This mode increments a register by one or two after the instruction has been carried out (,X+ and ,X++), or decrements the register by one or two before the instruction is carried out (,-Y and ,--Y).

Now we can briefly look at how indexed addressing can be used to perform some simple arithmetic on values in the index registers using the LEA (Load Effective Address) instruction. The normal arithmetic instructions will not work on the values in registers other than the accumulators. Although it is possible to transfer the contents of the index register into the D accumulator, perform the arithmetic and then transfer the result back, this is an awkward and slow procedure. The LEA instruction (which can be applied to the X, Y, S and U registers only) will perform any necessary address calculations and then load the effective address value. Normally the contents of an effective address would be loaded, so this is a useful alternative.

Let's take a look at an example. The instruction:

```
LEAX   -1,X
```

will calculate the effective address as the sum of −1 and the current contents of the X register. This address is then loaded back into X, effectively decrementing the value in that register. This is not the only use of this instruction; it could be used, for example, to carry out an address calculation once and save the result, rather than perform that same calculation a number of times.

It is also possible to do a certain amount of arithmetic on the X register using the ABX (Add B to X) instruction, which does an unsigned addition of the contents of B to the contents of X. However, this is not as generally useful as LEA.

SUBROUTINES

A subroutine is a self-contained section of code that is called from the main program (or another subroutine) to perform a specific task. Once that job has been done, control is automatically transferred back to the calling program at the instruction immediately following the original subroutine call. There are three main reasons for using subroutines:

1) To save writing the same piece of code more than once. It is more convenient to write an often used piece of code as a subroutine and call this when it is required.

2) So that a library of common routines can be built up, and then used in a number of different programs.

3) To break a program down into smaller, more manageable sections.

The most significant thing to remember about using subroutines in Assembly language is that both the calling program and the subroutine will be using the same registers. One of the most common errors in machine code programming occurs when, having stored a value in one of the registers, a program calls a subroutine and on its return finds that the contents of that register have been altered by the subroutine. Therefore, it is vital to know, and to document, the registers that a subroutine uses. It is particularly essential to save the contents of the registers being used when a subroutine is called, and restore those contents when control returns from the subroutine.

Later in the course we will look at how the stacks are used both as a convenient way of saving such data, and as a means of passing values and addresses (parameters) to the subroutine. For the moment, however, we shall assume that the subroutine uses the same data as the calling program (global variables) and any other values that it needs will actually be in the registers. A subroutine call is made by means of one of these instructions:

- BSR: Branch to SubRoutine
- JSR: Jump to SubRoutine

The BSR command causes a relative branch — it finds the subroutine at a certain offset from the current value of the program counter. This instruction is normally used for subroutines written as part of the program.

The JSR instruction calls a subroutine at a certain specified address. This would be used for a subroutine held in ROM, or for a library routine that always occupies the same position in memory — parts of the disk operating system, for example.

When the processor encounters a BSR or JSR instruction, the current value of the program counter is 'pushed' onto the system stack using the S (stack pointer) register.If your subroutine uses the S register for anything other than a further subroutine call, you must ensure that it gets restored to the correct value. The address of the subroutine is calculated (in the case of BSR) and loaded into the program counter. Thus, the next instruction to be accessed will be the first one of the subroutine. You must be sure, therefore, that the subroutine begins with an instruction and not a byte of data.

A subroutine must end with an RTS (ReTurn from Subroutine) instruction, the effect of which is to 'pull' the old value of the program counter back off the stack. Execution of the program will then continue from where it left off before the subroutine call.

The example program we give here is rather more complex than those we have given previously, but it can be made more manageable by the use of a subroutine. The program searches a table containing strings of unequal length, and extracts a value associated with one particular string. The strings are held in the normal way: beginning with a byte indicating the string's length, followed by the characters that make up the string, and ending with a 16-bit address associated with the string.

The end of the table is marked by a zero length string — in other words, there is a value of zero where the length byte should be. We shall assume that the address of the start of the table is held in $10, and the address of the string whose match we have to search for is held in $12. If the duplicate is found in the table, then the corresponding address is to be held in $14. If the string is not found, then $12 and $14 should be both set to zero.

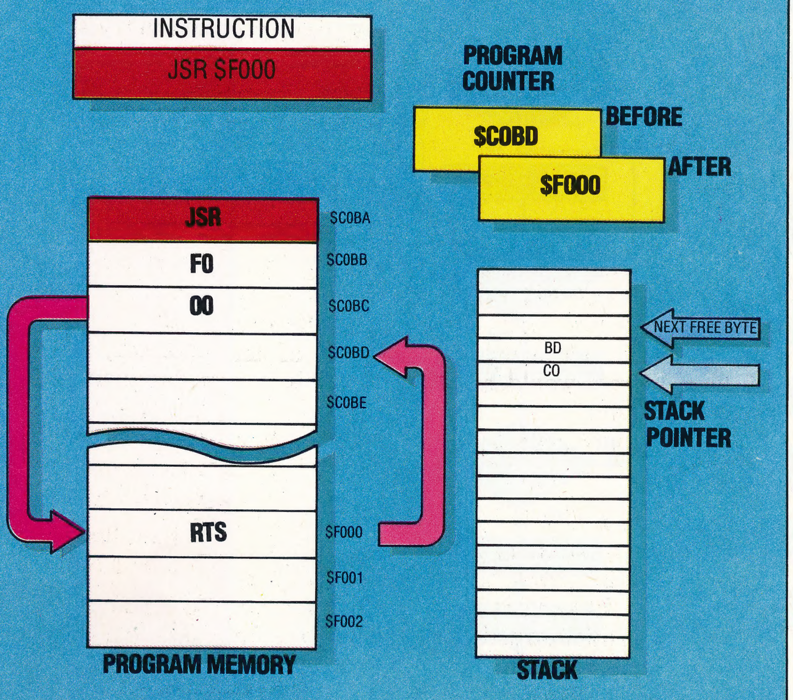
STRING-MATCHING

String-matching is a task that occurs in many situations — most notably in managing a BASIC interpreter's string variable accesses: each identifier (or variable name) must be replaced by the address in which the value of that variable is stored.

The problem divides easily into two parts: we must step through the table until either the string we are looking for is found or the end of the table is reached. At each stage in the search we must compare two strings (the one we are looking for and the one at the current position in the table) to see if they match.

This string comparison is an obvious candidate for a subroutine, because not only is it going to be used more than once in the program, but it also enables us to split up the problem into useful

Jumping Stack



The 'jump and return' implied in a subroutine call is managed by saving the present value of the program counter, replacing it with the subroutine call address, and, finally, restoring the program counter to its original state. The stack is an area of memory used by the processor for saving the return address, and the stack pointer is a 16-bit CPU register that always contains the address of the next free byte of stack space.

When JSR is encountered at, say, address $C0BA, the CPU automatically places $C0BD — the address of ABX, the instruction following JSR — in the program counter. When JSR is executed, the program counter contents are 'pushed' onto the stack by the CPU and $F000 is placed in the program counter. The subroutine is thus executed until RTS (ReTurn from Subroutine), is encountered, when $C0BD — the return address — is 'pulled' or 'popped' off the stack back into the program counter

KEVIN JONES

sections. It is also a good subroutine to have available for use in other programs.

The subroutine needs two data items from the calling program — namely, the addresses of the two strings to be compared. Since the subroutine has to step through the strings byte by byte, it is best that these two values are passed to the index registers, X and Y, where they will be needed. The subroutine must also pass back two values, one to indicate whether or not a match has been found, and the other to show the address itself in the case of a match.

TRUE OR FALSE

It is possible to pass a Boolean parameter (true or false) using one of the condition code register flags, but this requires an exact knowledge of the effect of each instruction on the flags. In our program we will pass values back to the calling routine as either $00 (all zeros) if the match is found, or $FF (all ones) if it is not.

To make the subroutine more generally useful, we won't pass back the actual address for a found match, but will instead leave the X register pointing to the address where the required address can be found. This has the additional advantage that the X register, by stepping byte by byte through the string, should end up containing this information automatically, anyway.

One final point: our program contains one new 6809 instruction. TST (TeST) has no effect on any register, but simply sets the flags according to the current value of the named register.

```
TABLE   EQU    $10
STRNG   EQU    $12
ADDRS   EQU    $14
        ORG    $1000      Start of main program
        LDU    TABLE      Start of table in U
LOOP1   LDA    ,U         Get length byte
        BEQ    NTFND1     GOTO end of table if zero
        ADDA   #3         Add one for length byte, two
                          for address to length of string,
                          giving length of table entry
        TFR    U,X        Put start of string in table into
                          X
        LEAU   A,U        Set U to point to next entry
        LDY    STRNG      Y points to start of search
                          string
        BSR    COMPAR     Compare the two strings
        TSTA              See if they are equal
        BEQ    FOUND1     If they are equal then GOTO
                          FOUND1
        BRA    LOOP1      ELSE get the next table entry
FOUND1  LDD    ,X         If found, X should be pointing
                          to the address we want
        BRA    FINSH1
NTFND1  LDD    #0         Address will be zero if not
                          found
FINSH1  STD    ADDRS      Save what we came for
        SWI               End of main program
COMPAR  LDB    ,X+        Start of subroutine
        CMPB   ,Y+        Get length bytes and point X
                          and Y to the first characters
        BNE    NOTEQ      If the strings are not the same
                          length then GOTO NOTEQ
LOOP2   LDA    ,X+        Get next character from table
                          string
        CMPA   ,Y+        Compare it with next character
                          from search string
        BNE    NOTEQ      Stop if they are not identical
        DECB              Else take one from position
                          pointer
        BGT    LOOP2      Get next character
        CLRA              Make A zero to show that the
                          strings are identical
        BRA    FINSH2
NOTEQ   LDA    #$FF       Ones if not equal
FINSH2  RTS               Back to calling program
        END
```

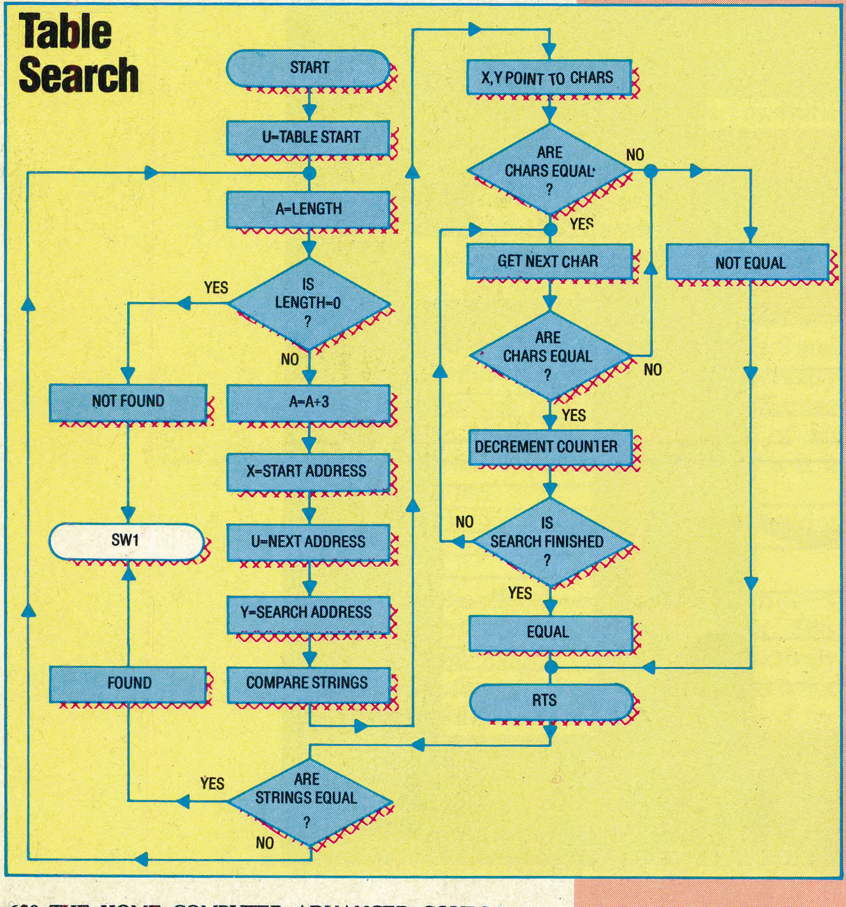


Table Search

DATABASE

Here, courtesy of Zilog Inc., we produce the second part of the programmers' Z80 reference card.

16-Bit Load Group

Mnemonic	Symbolic Operation	S	Z		H		P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
LD dd, nn	dd ← nn	•	•	X	•	X	•	•	•	00 dd0 001 ← n → ← n →		3	3	10	dd Pair 00 BC 01 DE 10 HL 11 SP
LD IX, nn	IX ← nn	•	•	X	•	X	•	•	•	11 011 101 00 100 001 ← n → ← n →	DD 21	4	4	14	
LD IY, nn	IY ← nn	•	•	X	•	X	•	•	•	11 111 101 00 100 001 ← n → ← n →	FD 21	4	4	14	
LD HL, (nn)	H ← (nn + 1) L ← (nn)	•	•	X	•	X	•	•	•	00 101 010 ← n → ← n →	2A	3	5	16	
LD dd, (nn)	dd_H ← (nn + 1) dd_L ← (nn)	•	•	X	•	X	•	•	•	11 101 101 01 dd1 011 ← n → ← n →	ED	4	6	20	
LD IX, (nn)	IX_H ← (nn + 1) IX_L ← (nn)	•	•	X	•	X	•	•	•	11 011 101 00 101 010 ← n → ← n →	DD 2A	4	6	20	
LD IY, (nn)	IY_H ← (nn + 1) IY_L ← (nn)	•	•	X	•	X	•	•	•	11 111 101 00 101 010 ← n → ← n →	FD 2A	4	6	20	
LD (nn), HL	(nn + 1) ← H (nn) ← L	•	•	X	•	X	•	•	•	00 100 010 ← n → ← n →	22	3	5	16	
LD (nn), dd	(nn + 1) ← dd_H (nn) ← dd_L	•	•	X	•	X	•	•	•	11 101 101 01 dd0 011 ← n → ← n →	ED	4	6	20	
LD (nn), IX	(nn + 1) ← IX_H (nn) ← IX_L	•	•	X	•	X	•	•	•	11 011 101 00 100 010 ← n → ← n →	DD 22	4	6	20	
LD (nn), IY	(nn + 1) ← IY_H (nn) ← IY_L	•	•	X	•	X	•	•	•	11 111 101 00 100 010 ← n → ← n →	FD 22	4	6	20	
LD SP, HL	SP ← HL	•	•	X	•	X	•	•	•	11 111 001	F9	1	1	6	
LD SP, IX	SP ← IX	•	•	X	•	X	•	•	•	11 011 101 11 111 001	DD F9	2	2	10	
LD SP, IY	SP ← IY	•	•	X	•	X	•	•	•	11 111 101 11 111 001	FD F9	2	2	10	
PUSH qq	(SP – 2) ← qq_L (SP – 1) ← qq_H SP → SP – 2	•	•	X	•	X	•	•	•	11 qq0 101		1	3	11	qq Pair 00 BC 01 DE 10 HL 11 AF
PUSH IX	(SP – 2) ← IX_L (SP – 1) ← IX_H SP → SP – 2	•	•	X	•	X	•	•	•	11 011 101 11 100 101	DD E5	2	4	15	
PUSH IY	(SP – 2) ← IY_L (SP – 1) ← IY_H SP → SP – 2	•	•	X	•	X	•	•	•	11 111 101 11 100 101	FD E5	2	4	15	
POP qq	qq_H ← (SP + 1) qq_L ← (SP) SP → SP + 2	•	•	X	•	X	•	•	•	11 qq0 001		1	3	10	
POP IX	IX_H ← (SP + 1) IX_L ← (SP) SP → SP + 2	•	•	X	•	X	•	•	•	11 011 101 11 100 001	DD E1	2	4	14	
POP IY	IY_H ← (SP + 1) IY_L ← (SP) SP → SP + 2	•	•	X	•	X	•	•	•	11 111 101 11 100 001	FD E1	2	4	14	

NOTES dd is any of the register pairs BC, DE, HL, SP.
qq is any of the register pairs AF, BC, DE, HL.
$(PAIR)_H$, $(PAIR)_L$ refer to high order and low order eight bits of the register pair respectively.
e.g., BC_L = C, AF_H = A

Flag Notation • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
↕ = flag is affected according to the result of the operation.

			SOURCE									
			REGISTER							IMM. EXT.	EXT. ADDR.	REG. INDIR.
			AF	BC	DE	HL	SP	IX	IY	nn	(nn)	(SP)
DESTINATION	REGISTER	AF										F1
		BC								01 n n	ED 4B n n	C1
		DE								11 n n	ED 5B n n	D1
		HL								21 n n	2A n n	E1
		SP				F9		DD F9	FD F9	31 n n	ED 7B n n	
		IX								DD 21 n n	DD 2A n n	DD E1
		IY								FD 21 n n	FD 2A n n	FD E1
	EXTERNAL ADDRESS	(nn)		ED 43 n n	ED 53 n n	22 n n	ED 73 n n	DD 22 n n	FD 22 n n			
PUSH INSTRUCTIONS	REGISTER IND.	(SP)	F5	C5	D5	E5		DD E5	FD E5			

NOTE: The Push & Pop Instructions adjust the SP after every execution.